HTML5

ARKAITZ GARRO

SOBRE ESTA EDICIÓN

HTML5

Publication date: 28/01/2014

This book was published with *easybook v5.0-DEV*, a free and open-source book publishing application developed by Javier Eguiluz using several Symfony components.

LICENCIA

Esta obra se publica bajo la licencia *Creative Commons Reconocimiento - No Comercial - Compartir Igual 3.0*, cuyos detalles puedes consultar en http://creativecommons.org/licenses/by-nc-sa/3.0/es/.

Puedes copiar, distribuir y comunicar públicamente la obra, incluso transformándola, siempre que cumplas todas las condiciones siguientes:

Reconocimiento: debes reconocer siempre la autoría de la obra original, indicando tanto el nombre del autor (Arkaitz Garro) como el nombre del sitio donde se publicó originalmente www.arkaitzgarro.com. Este reconocimiento no debe hacerse de una manera que sugiera que el autor o el sitio apoyan el uso que haces de su obra.

No comercial: no puedes utilizar esta obra con fines comerciales de ningún tipo. Entre otros, no puedes vender esta obra bajo ningún concepto y tampoco puedes publicar estos contenidos en sitios web que incluyan publicidad de cualquier tipo.

Compartir igual: si alteras o transformas esta obra o si realizas una obra derivada, debes compartir tu trabajo obligatoriamente bajo esta misma licencia.

CAPÍTULO 1 ¿QUÉ ES HTML5?

HTML5 (*HyperText Markup Language*, versión 5) es la quinta revisión del lenguaje HTML. Esta nueva versión (aún en desarrollo), y en conjunto con CSS3, define los nuevos estándares de desarrollo web, rediseñando el código para resolver problemas y actualizándolo así a nuevas necesidades. No se limita solo a crear nuevas etiquetas o atributos, sino que incorpora muchas características nuevas y proporciona una plataforma de desarrollo de complejas aplicaciones web (mediante los APIs).

HTML5 está destinado a sustituir no sólo HTML 4, sino también XHTML 1 y DOM Nivel 2. Esta versión nos permite una mayor interacción entre nuestras páginas web y el contenido media (video, audio, entre otros) así como una mayor facilidad a la hora de codificar nuestro diseño básico.

Algunas de las nuevas características de HTML5 serían:

- Nuevas etiquetas semánticas para estructurar los documentos HTML, destinadas a remplazar la necesidad de tener una etiqueta `<div>` que identifique cada bloque de la página.
- Los nuevos elementos multimedia como `<audio>` y `<video>`.
- La integración de gráficos vectoriales escalables (SVG) en sustitución de los genéricos `<object>`, y un nuevo elemento `<canvas>` que nos permite *dibujar* en él.
- El cambio, redefinición o estandarización de algunos elementos, como `<a>`, `<cite>` o `<menu>`.
- `MathML` para fórmulas matemáticas.
- Almacenamiento local en el lado del cliente.
- Y otros muchos nuevos APIs que veremos a lo largo de los siguientes capítulos.

1.1 ESPECIFICACIÓN OFICIAL

El organismo W3C elabora las normas a seguir para la creación de las páginas HTML5. Sin embargo, no es necesario conocer todas estas especificaciones, escritas es un lenguaje bastante formal, para diseñar páginas con este lenguaje. Las normas oficiales están escritas en inglés y se pueden consultar de forma gratuita en las siguientes direcciones:

- Especificación recomendada como candidata para HTML5
- Borrador para la especificación oficial de HTML 5.1

1.2 5 TIPS

1.2.1 HTML5 ES API

Se puede pensar en HTML sólo como nuevas etiquetas y geolocalización. Pero esta no es más que una pequeña parte del estándar que define HTML5. La especificación de HTML5 define también cómo esas etiquetas interactúan con JavaScript, a través del Modelo de Objetos de Documento (DOM). HTML5 no es únicamente definir una etiqueta como `<video>`, también existe su correspondiente API para objetos de vídeo en el DOM. Se puede utilizar esta API para detectar el soporte para diferentes formatos de vídeo, reproducir el vídeo, hacer una pausa, silenciar el audio, realizar un seguimiento de la cantidad de vídeo que se ha descargado, y todo lo que necesita para crear una completa experiencia de usuario alrededor de la etiqueta en sí.

1.2.2 NO HAY QUE TIRAR NADA A LA BASURA

Se puede amar, o se puede odiar, pero no se puede negar que HTML 4 es el formato de marcado más exitoso de la historia. HTML5 se basa en ese éxito. No es necesario volver a aprender cosas que ya se conocen. Si la aplicación web que funcionaba ayer en HTML 4, hoy funcionará en HTML5.

Ahora, si lo que se desea es mejorar las aplicaciones web, este es el lugar correcto. He aquí un ejemplo concreto: HTML5 soporta todos los controles de formulario de HTML 4, pero también incluye nuevos controles de entrada. Algunos de estos son funcionalidades esperadas durante mucho tiempo, como reguladores y selectores de fecha, mientras que otros son más sutiles. Por ejemplo, el tipo de entrada de correo electrónico se parece a un cuadro de texto, pero los navegadores móviles personalizar su teclado en pantalla para que sea más fácil de escribir direcciones de correo electrónico. Los navegadores más antiguos que no son compatibles con el tipo de entrada de correo electrónico será tratado como un campo de texto normal, y el formulario sigue funcionando sin ningún cambio en las etiquetas o *hacks* de JavaScript.

1.2.3 FÁCIL DE COMENZAR

"Actualizar" a HTML5 puede ser tan simple como cambiar su tipo de documento. El tipo de documento debe estar en la primera línea de cada página HTML. Las versiones anteriores de HTML definen un montón de *doctypes*, y elegir el más adecuado puede ser difícil. En HTML5, sólo hay un tipo de documento:

```
<!DOCTYPE html>
```

La actualización al doctype HTML5 no rompe el marcado existente, ya que los elementos obsoletos previamente definidas en HTML 4 todavía se representará en HTML5. Pero le permitirá usar (y validar) nuevos elementos semánticos como `<article>`, `<section>`, `<header>` y `<footer>`.

1.2.4 ¡YA FUNCIONA!

Si se quiere dibujar en un lienzo, reproducir vídeo, diseñar mejores formas, o construir aplicaciones web que funcionan *offline*, nos encontramos con que HTML5 ya está bien soportado. Firefox, Safari, Chrome, Opera y los navegadores móviles ya son compatibles con canvas, video, la geolocalización, el almacenamiento local, y más funcionalidades. Incluso Microsoft (raramente conocido por el soporte de estándares) soporta la mayoría de las características de HTML5 en Internet Explorer 9.

1.2.5 HA VENIDO PARA QUEDARSE

Tim Berners-Lee inventó la World Wide Web a principios de 1990. Más tarde fundó el W3C para que actuase como administrador único de los estándares web, lo que venido haciendo durante más de 15 años. Esto es lo que el W3C tenía que decir sobre el futuro de los estándares web, en julio de 2009:

> *Hoy, el director anuncia que cuando el XHTML 2 expire en la fecha prevista a finales de 2009, no será renovado. De este modo, y mediante el aumento de los recursos en el Grupo de Trabajo de HTML, el W3C espera acelerar el progreso de HTML5 y aclarar la posición del W3C sobre el futuro de HTML.*

En septiembre de 2012, el W3C propuso un plan para crear una primera especificación de HTML5 a finales de 2014, y una nueva especificación final de HTML 5.1 a finales 2016. Al igual que ocurre en la especificación de CSS3, en HTML5 se ha optado por modularizar la especificación, creando grupos de trabajo que trabajan de forma separada en diferentes aspectos del estándar. Algunas de las especificaciones sobre las que se está trabajando:

- HTML Microdata - HTML WG
- HTML Canvas 2D Context - HTML WG
- HTML5 Web Messaging - Web Apps WG
- Web Workers - Web Apps WG
- Web Storage - Web Apps WG
- The WebSocket API - Web Apps WG
- The WebSocket Protocol - IETF HyBi WG
- Server-Sent Events - Web Apps WG
- WebRTC - WebRTC WG
- WebVTT - W3C Web Media Text Tracks CG

ÍNDICE DE CONTENIDOS

CAPÍTULO 2 SEMÁNTICA

Una de las novedades que hemos mencionado anteriormente son las etiquetas que se han introducido en HTML5. Existen más de 30 nuevas etiquetas semánticas que pueden ser utilizadas en nuestras páginas estáticas. Estas nuevas etiquetas se podrían clasificar en dos grupos:

- Etiquetas que extienden a las actuales, como `<video>`, `<audio>` o `<canvas>`, y que además añaden nuevas funcionalidades a los documentos HTML, que podemos controlar desde JavaScript y
- etiquetas que componen la web semántica, es decir, que no proponen nuevas funcionalidades pero sirven para estructurar sitios web, y añadir un significado concreto, más allá de las etiquetas generales como `<div>`.

En este capítulo, veremos como transformar nuestra estructura actual de marcado basada en `<div>`, a una estructura que utiliza las nuevas etiquetas estructurales como `<nav>`, `<header>`, `<footer>`, `<aside>`, o `<article>`.

2.1 CABECERA DEL DOCUMENTO

Además de las nuevas etiquetas introducidas por HTML5 (que veremos más adelante), el nuevo estándar propone pequeñas mejoras que podemos aplicar en la definición de nuestros documentos, en concreto en la cabecera de los mismos.

2.1.1 DOCTYPE

El estándar XHTML deriva de XML, por lo que comparte con él muchas de sus normas y sintaxis. Uno de los conceptos fundamentales de XML es la utilización del DTD o Document Type Definition ("Definición del Tipo de Documento"). El estándar XHTML define el DTD que deben seguir las páginas y documentos XHTML. En este documento se definen las etiquetas que se pueden utilizar, los atributos de cada etiqueta y el tipo de valores que puede tener cada atributo.

```
<!DOCTYPE html
      PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Ésta es una de las 15 declaraciones posibles declaradas en los estándares HTML4 y XHTML. En HTML5 se reduce la definición del tipo de documento a una única posibilidad, por lo que no tenemos que preocuparnos de elegir el tipo de documento correcto:

```
<!DOCTYPE html>
```

2.1.2 ELEMENTO RAÍZ HTML

En todo documento HTML, su elemento raíz o nodo superior siempre es la etiqueta `<html>`. Hasta ahora, este elemento raíz se definía de la siguiente manera:

```
<html xmlns=http://www.w3.org/1999/xhtml
   lang="en"
   xml:lang="en">
```

No hay ningún problema en mantener esta sintaxis. Si se desea, se puede conservar, ya que es válido en HTML5. Sin embargo, algunas de sus partes ya no son necesarias, por lo que podemos eliminarlas.

El primer elemento del que podemos prescindir es el atributo `xmlns`. Se trata de una herencia de XHTML 1.0, que dice que los elementos de esta página están en el espacio de nombres XHTML, `http://www.w3.org/1999/xhtml`. Sin embargo, los elementos de HTML5 están siempre en este espacio de nombres, por lo que ya no es necesario declararlo explícitamente. Eliminar el atributo `xmlns` nos deja con este elemento de la siguiente manera:

```
<html lang="es" xml:lang="en">
```

En este caso ocurre lo mismo con el atributo `xml:lang`, es una herencia de XHTML que podemos eliminar, quedando finalmente la etiqueta de la siguiente manera:

```
<html lang="en">
```

2.1.3 ELEMENTO HEAD

El primer hijo del elemento raíz es generalmente el elemento `head`. El elemento `head` contiene los metadatos que aportan información extra sobre la página, como su título, descripción, autor, etc. Además, puede incluir referencias externas a contenidos necesarios para que el documento se muestre y comporte de manera correcta (como hojas de estilos o *scripts*). Este elemento ha sufrido pequeñas variaciones, pero que debemos tener en cuenta:

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>My Weblog</title>
    <link rel="stylesheet" type="text/css" href="style-original.css" />
    <link rel="alternate" type="application/atom+xml"
                          title="My Weblog feed"
                          href="/feed/" />
    <link rel="search" type="application/opensearchdescription+xml"
```

```
                          title="My Weblog search"
                          href="opensearch.xml"  />
    <link rel="shortcut icon" href="/favicon.ico" />
</head>
```

2.1.3.1 CODIFICACIÓN DEL CONTENIDO

Cuando se piensa en "texto", probablemente nos venga a la cabeza una definición de "caracteres y símbolos que veo en la pantalla de mi ordenador". Pero realmente se tratan de bits y bytes. Cada cadena de caracteres que se muestra en la pantalla, se almacena con una codificación de caracteres en particular. Hay cientos de codificaciones de caracteres diferentes, algunos optimizado para ciertos idiomas como el ruso, el chino o inglés, y otros que se pueden utilizar para múltiples idiomas. En términos generales, la codificación de caracteres proporciona una correspondencia entre lo que se muestra en la pantalla y lo que un equipo realmente almacena en la memoria y en el disco.

Se puede pensar en la codificación de caracteres como una especie de clave de descifrado del texto. Cuando accedemos a una secuencia de bytes, y decidimos que es "texto", lo que necesitamos saber es qué codificación de caracteres se utiliza para que pueda decodificar los bytes en caracteres y mostrarlos (o transformarlos) de manera correcta.

Lo ideal es establecer esta codificación en el servidor, indicando el tipo en las cabeceras de respuesta:

```
Content-Type: text/html; charset="utf-8"
```

Por desgracia, no siempre podemos tener el control sobre la configuración de un servidor HTTP. Por ejemplo, en la plataforma *Blogger*, el contenido es proporcionado por personas, pero los servidores son administrados por Google, por lo que estamos supeditados a su configuración. Aún así, HTML 4 proporciona una manera de especificar la codificación de caracteres en el documento HTML:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

El encabezado HTTP es el método principal, y anula la etiqueta `<meta>` si está presente. Pero no todo el mundo puede establecer encabezados HTTP, por lo que la etiqueta `<meta>` todavía tiene sentido. En el caso de HTML5, es aún más sencillo definir esta etiqueta *meta*:

```
<meta charset="utf-8">
```

Debemos acostumbrarnos a especificar la codificación de nuestros documentos, aunque no vayamos a utilizar caracteres especiales o nuestro documentos no se presente en otros idiomas. Si no lo hacemos, podemos exponernos a problemas de seguridad.

2.1.3.2 LINKS

Dentro del elemento `head`, las etiquetas `<link>` son una manera de acceder o declarar contenido externo al documento actual, que puede cumplir distintos objetivos:

- Es una hoja de estilo contiene las reglas CSS que su navegador debe aplicar al presente documento.
- Es un feed que contiene el mismo contenido que esta página, pero en un formato estándar (RSS).
- Es una traducción de esta página en otro idioma.
- Es el mismo contenido que esta página, pero en formato PDF.
- Es el próximo capítulo de un libro en línea de la cual esta página es también una parte.

En HTML5, se separan estas relaciones de enlace en dos categorías:

- Enlaces a recursos externos que se van a utilizar para mejorar el documento actual,
- y enlaces de hipervínculos que son enlaces a otros documentos.

El tipo de relación más utilizado (literalmente) es el siguiente:

```
<link rel="stylesheet" href="style-original.css" type="text/css" />
```

Esta relación es utilizada para indicar el archivo donde se almacenan las reglas CSS que se desean aplicar al documento. Una pequeña optimización que se puede hacer en HTML5 es eliminar el atributo `type`. Sólo hay un lenguaje de estilo para la web, CSS, así que ese es el valor predeterminado para el atributo `type`:

```
<link rel="stylesheet" href="style-original.css" />
```

2.2 NUEVAS ETIQUETAS SEMÁNTICAS

En 2004, Ian Hickson, el autor de la especificación de HTML5, analizó 1.000.000.000 de páginas web utilizando el motor de Google, intentando identificar la manera en la que la *web real* estaba construida. Uno de los resultados de este análisis, fue la publicación de una lista con los nombres de *clases* más utilizados. Este estudio revela que los desarrolladores utilizan *clases* o *IDs* comunes para estructurar los documentos. Esto llevó a considerar que quizás fuese una buena idea crear etiquetas concretas para reflejar estas estructuras.

Este tipo de etiquetas que componen la web semántica nos sirven para que cualquier mecanismo automático (un navegador, un motor de búsqueda, un lector de *feeds*...) que lea un sitio web sepa con exactitud qué partes de su contenido corresponden a cada una de las partes típicas de un sitio. Observando esas etiquetas semánticas estructurales,

cualquier sistema podrá procesar la página y saber cómo está estructurada. Veamos algunas de estas etiquetas que introduce HTML5 en este sentido.

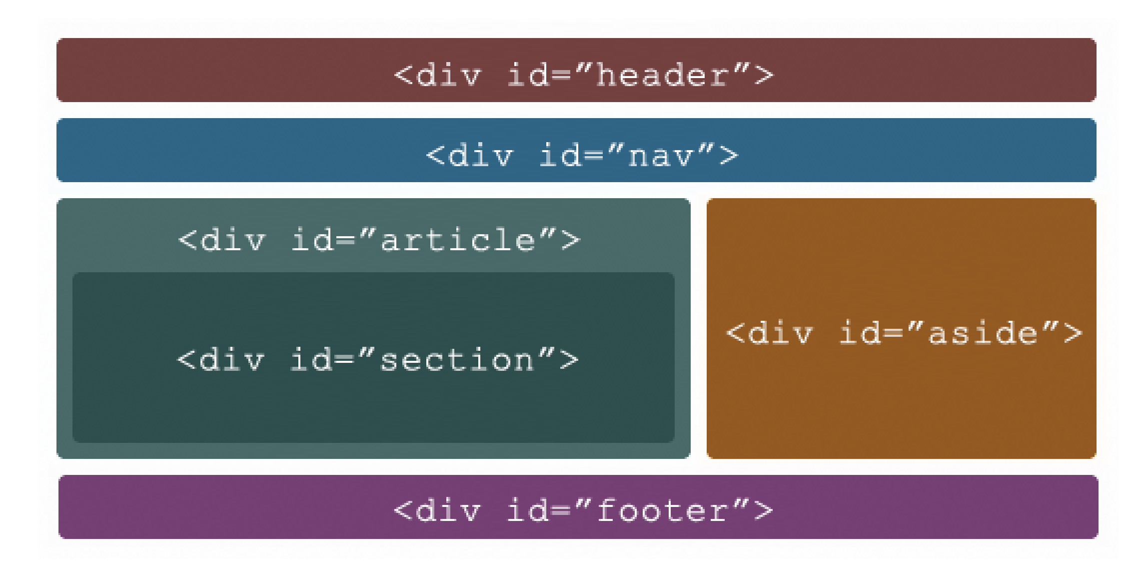
- `<section></section>`: se utiliza para representar una sección "general" dentro de un documento o aplicación, como un capítulo de un libro. Puede contener subsecciones y si lo acompañamos de `h1-h6` podemos estructurar mejor toda la página creando jerarquías del contenido, algo muy favorable para el buen posicionamiento web.
- `<article></article>`: representa un componente de una página que consiste en una composición autónoma en un documento, página, aplicación, o sitio web con la intención de que pueda ser reutilizado y repetido. Podría utilizarse en los artículos de los foros, una revista o el artículo de periódico, una entrada de un blog, un comentario escrito por un usuario, un widget interactivo o cualquier otro artículo independiente de contenido. Cuando los elementos de `<article>` son anidados, los elementos interiores representan los artículos que en principio son relacionados con el contenido del artículo externo. Por ejemplo, un artículo de un blog que permite comentarios de usuario, dichos comentarios se podrían representar con `<article>`.
- `<aside></aside>`: representa una sección de la página que abarca un contenido relacionado con el contenido que lo rodea, por lo que se le puede considerar un contenido independiente. Este elemento puede utilizarse para efectos tipográficos, barras laterales, elementos publicitarios, para grupos de elementos de la navegación, u otro contenido que se considere separado del contenido principal de la página.
- `<header></header>`: representa un grupo de artículos introductorios o de navegación. Está destinado a contener por lo general la cabecera de la sección (un elemento `h1-h6` o un elemento `hgroup`), pero no es necesario.
- `<nav></nav>`: representa una sección de una página que enlaza a otras páginas o a otras partes dentro de la página. No todos los grupos de enlaces en una página necesita estar en un elemento `nav`, sólo las secciones que constan de bloques de navegación principales son apropiados para el elemento de navegación.
- `<footer></footer>`: representa el pie de una sección, con información acerca de la página/sección que poco tiene que ver con el contenido de la página, como el autor, el copyright o el año.
- `<hgroup></hgroup>`: representa el encabezado de una sección. El elemento se utiliza para agrupar un conjunto de elementos `h1-h6` cuando el título tiene varios niveles, tales como subtítulos o títulos alternativos.
- `<time>`: representa o bien una hora (en formato de 24 horas), o una fecha precisa en el calendario gregoriano (en formato ISO), opcionalmente con un tiempo y un desplazamiento de zona horaria.

2.3 ESTRUCTURA DE UN DOCUMENTO HTML5

Como hemos visto con las nuevas etiquetas semánticas introducidas en HTML5, éstas aportan un significado concreto al documento que estamos definiendo, y por lo tanto, afectan de manera directa a la forma en la estructuramos el contenido. Vamos a ver como podemos *convertir* nuestra actual página con las nuevas etiquetas introducidas en HTML5.

2.3.1 ESTRUCTURA TRADICIONAL DE UN DOCUMENTO EN HTML 4

El siguiente código muestra una estructura "clásica" de documento HTML, donde los diferentes contenidos de la web se encuentran agrupados por etiquetas `<div>`. Por sí mismas, estas etiquetas no aportan ningún tipo de significado, y el atributo `id` tampoco se lo proporciona. Si cambiamos `<div id="header">` por `<div id="whatever">`, el significado sigue siendo el mismo, *ninguno*.

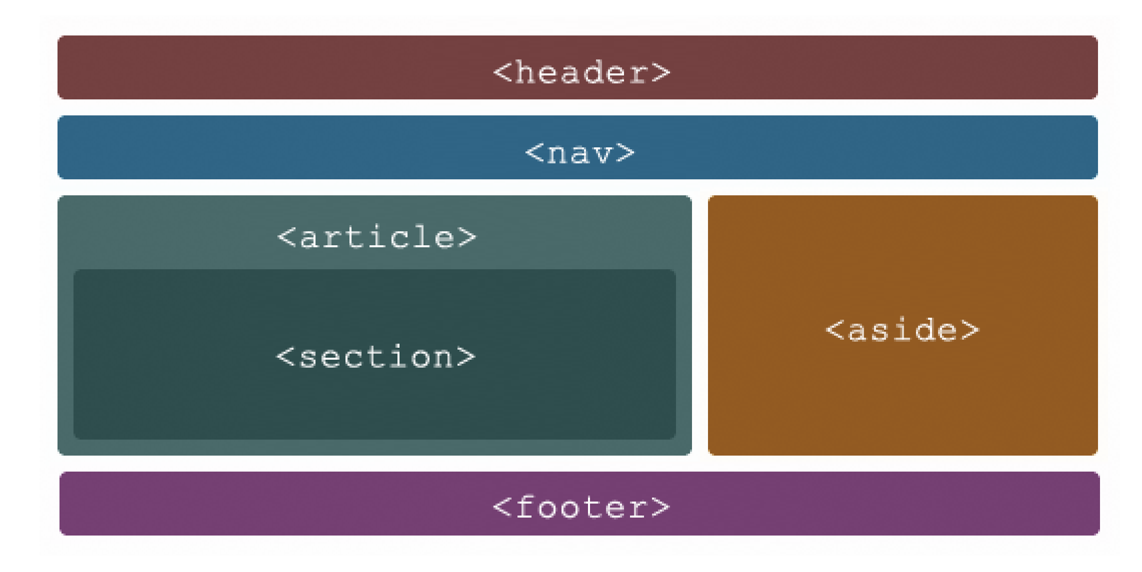



```
<DOCTYPE html PUBLIC"-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<body>
    <div id="header">
        <a href="/"><img src=logo.png alt="home"></a>
        <h1>My Weblog</h1>
        <p class="tagline">
            A lot of effort went into making this effortless.
        </p>
    </div>
    <div id="nav">
        <ul>
            <li><a href="#">home</a></li>
            <li><a href="#">blog</a></li>
            <li><a href="#">gallery</a></li>
            <li><a href="#">about</a></li>
        </ul>
```

```
    </div>
    <div class="articles">
        <div class="article">
            <p class="post-date">October 22, 2009</p>
            <h2>
                <a href="#" title="link to this post">Travel day</a>
            </h2>
            <div class="content">
                Content goes here...
            </div>
            <div class="comments">
                <p><a href="#">3 comments</a></p>
            </div>
        </div>
    </div>
    <div class="aside">
        <div class="related"></div>
        <div class="related"></div>
        <div class="related"></div>
    </div>
    <div id="footer">
        <p>&#167;</p>
        <p>&#169; 2013&#8211;9 <a href="#">Arkaitz Garro</a></p>
    </div>
</body>
</html>
```

2.3.2 ESTRUCTURA TRADICIONAL DE UN DOCUMENTO EN HTML5

Veamos como podemos añadir un significado a este documento, únicamente aplicando las nuevas etiquetas semánticas incluidas en HTML5.

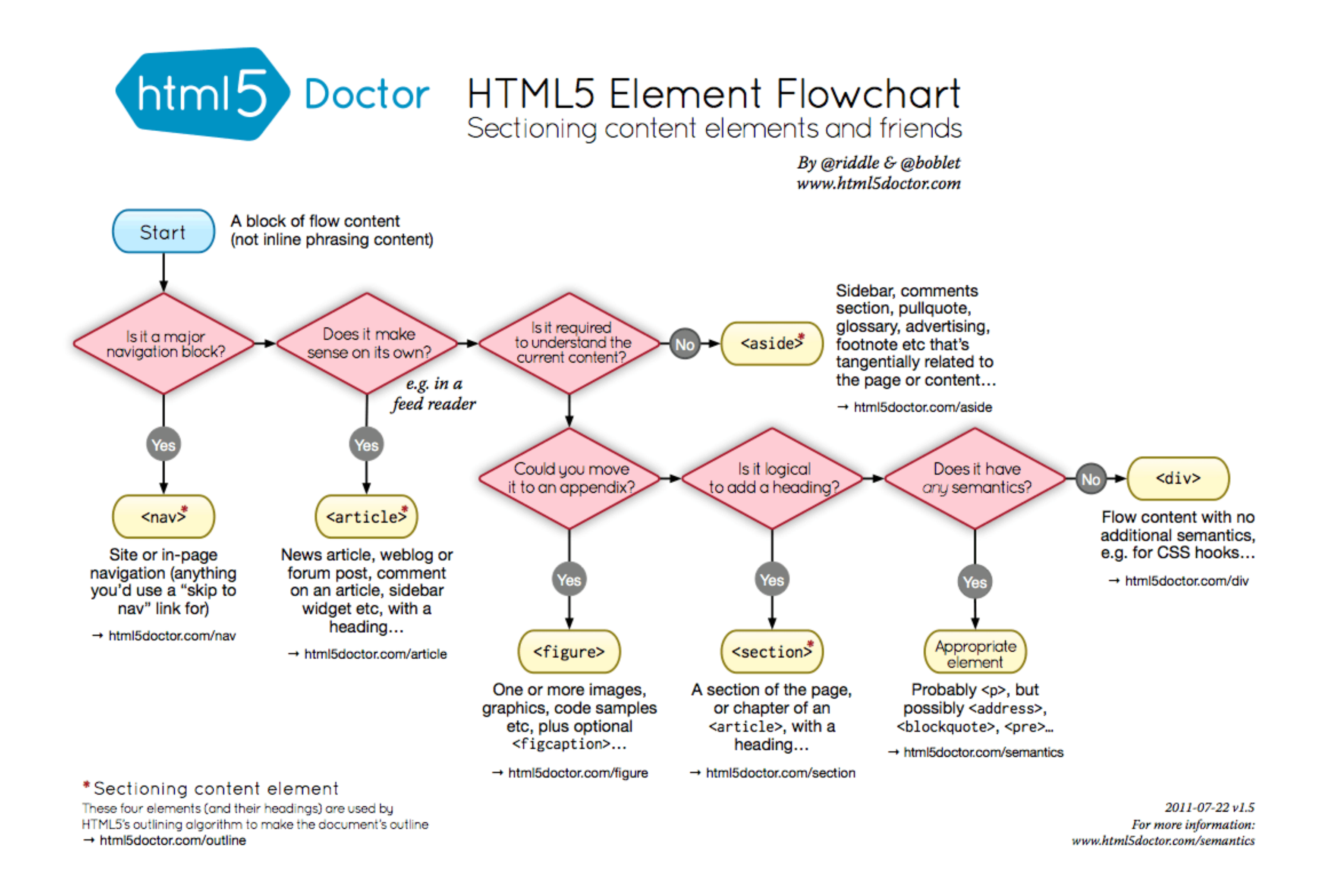



```
<!DOCTYPE html>
<html>
```

```html
<body>
    <header>
        <a href="/"><img src=logo.png alt="home"></a>
        <hgroup>
            <h1>Title</h1>
            <h2 class="tagline">
                A lot of effort went into making this effortless.
            </h2>
        </hgroup>
    </header>
    <nav>
        <ul>
            <li><a href="#">home</a></li>
            <li><a href="#">blog</a></li>
            <li><a href="#">gallery</a></li>
            <li><a href="#">about</a></li>
        </ul>
    </nav>
    <section class="articles">
        <article>
            <time datetime="2009-10-22">October 22, 2009</time>
            <h2>
                <a href="#" title="link to this post">Travel day</a>
            </h2>
            <div class="content">
                Content goes here...
            </div>
            <section class="comments">
                <p><a href="#">3 comments</a></p>
            </section>
        </article>
    </section>
    <aside>
        <div class="related"></div>
        <div class="related"></div>
        <div class="related"></div>
    </aside>
    <footer>
        <p>&#167;</p>
        <p>&#169; 2013&#8211;9 <a href="#">Arkaitz Garro</a></p>
    </footer>
</body>
</html>
```

Partiendo de la anterior estructura, parece evidente las nuevas etiquetas que debemos utilizar. Esto no siempre es así, y cuando estructuramos contenidos de mucho mayor alcance, lo normal es que nos surjan dudas. Un sencillo algoritmo que nos puede ayudar

en la correcta selección de etiquetas, es el que proponen en HTML5 Doctor.



2.4 USO DE LAS NUEVAS ETIQUETAS SEMÁNTICAS

2.4.1 `<HEADER>`

Según la especificación, un elemento `<header>` representa lo siguiente:

> *Un grupo de navegación o contenido introductorio. Un elemento header normalmente contiene una sección de encabezado (un elemento h1-h6 o un elemento hgroup), pero puede contener otro tipo de elementos, como una tabla de contenidos, un formulario de búsqueda o cualquier logo importante.*

En nuestro ejemplo, y en la mayoría de los sitios web, la cabecera contiene los primeros elementos de la página. Tradicionalmente el título de la web, su logo, enlaces para volver al inicio... De la manera más simple, nuestra cabecera quedaría de esta forma:

```
<header>
    <a href="/"><img src=logo.png alt="home"></a>
    <h1>Title</h1>
</header>
```

También es muy común que los sitios web muestren un lema o subtítulo bajo el título

principal. Para dar mayor importancia a este subtítulo, y relacionarlo de alguna manera con el título principal de la web, es posible agrupar los dos titulares bajo un elemento `<hgroup>`.

```
<header>
    <a href="/"><img src=logo.png alt="home"></a>
    <hgroup>
        <h1>Title</h1>
        <h2 class="tagline">
            A lot of effort went into making this effortless.
        </h2>
    </hgroup>
</header>
```

2.4.2 `<NAV>`

Según la especificación, un elemento `<nav>` representa lo siguiente:

> *El elemento `<nav>` representa una sección de una página que enlaza con otras páginas o partes de la misma página: una sección con enlaces de navegación.*

El elemento `<nav>` ha sido diseñado para identificar la navegación de un sitio web. La navegación se define como un conjunto de enlaces que hacen referencia a las secciones de una página o un sitio, pero no todos los enlaces son candidatos de pertenecer a un elemento `<nav>`: una lista de enlaces a patrocinadores o los resultados de una búsqueda, no forman parte de la navegación principal, sino que corresponden con el contenido de la página.

Como ocurre con los elementos `<header>`, `<footer>` y el resto de nuevas etiquetas, no estamos obligados a utilizar un único elemento `<nav>` en toda la página. Es posible que tengamos una navegación principal en la cabecera de la página, una tabla de contenidos o enlaces en el pie de la página, que apuntan a contenidos secundarios. Todos ellos son candidatos a pertenecer a un elemento `<nav>`.

```
<nav>
    <ul>
        <li><a href="#">home</a></li>
        <li><a href="#">blog</a></li>
        <li><a href="#">gallery</a></li>
        <li><a href="#">about</a></li>
    </ul>
</nav>
```

2.4.3 `<FOOTER>`

Según la especificación, un elemento `<footer>` representa lo siguiente:

> *Representa el pie de una sección. Un pie tradicionalmente contiene información acerca de su sección, como quién escribió el contenido, enlaces relacionados, copyright y similares.*

Al igual que ocurre con el elemento `<nav>`, podemos tener tantos elementos `<footer>` como sea necesario. Lo normal es que nuestro sitio web disponga de al menos un pie principal, que contiene los avisos legales (privacidad, condiciones del servicio, copyright...), mapa del sitio web, accesibilidad, contacto y otros muchos enlaces que pueden ir incluidos en un elemento `<nav>`.

2.4.4 `<ARTICLE>`

Según la especificación, un elemento `<article>` representa lo siguiente:

> *Este elemento representa un contenido completo, auto-contenido en un documento, página, aplicación o sitio web, que es, en principio, independiente de ser distribuido y reutilizado, por ejemplo en un RSS. Puede ser un post de un foro, un artículo de un periódico o revista, una entrada de un blog, un comentario de un usuario, un widget o cualquier otro elemento independiente.*
>
> *Cuando los artículos están anidados, los artículos interiores representan contenido que en principio está relacionado con el artículo que los contiene. Por ejemplo, una entrada de un blog puede aceptar comentarios de usuarios, que están incluidos dentro del contenido principal y relacionados con el mismo.*

Por lo tanto, la etiqueta `<article>` se utiliza para encapsular contenido, que tiene significado en sí mismo, y que puede ser distribuido y reutilizado en otros formatos de datos. No nos referimos únicamente a contenidos clásicos de texto, sino que incluso un contenido multimedia con su transcripción, un mapa o email pueden ser totalmente válidos para ser incluidos en una etiqueta `<article>`.

```
<section>
    <h1>Comments</h1>
    <article id="c1">
        <footer>
            <p>Posted by: <span>George Washington</span></p>
            <p><time datetime="2009-10-10">15 minutes ago</time></p>
        </footer>
        <p>Yeah! Especially when talking about your lobbyist friends!</p>
    </article>
    <article id="c2">
        <footer>
            <p>Posted by: <span itemprop="name">George Hammond</span></p>
            <p><time datetime="2009-10-10">5 minutes ago</time></p>
        </footer>
        <p>Hey, you have the same first name as me.</p>
```

```
    </article>
</section>
```

2.4.5 `<SECTION>`

A diferencia del elemento `<article>`, este elemento es utilizado para dividir el documento (o artículos) en diferentes áreas, o como su propio nombre indica, en secciones. Según la especificación, un elemento `<section>` representa lo siguiente:

> *Representa una sección genérica de un documento o aplicación. Una sección, en este contexto, es un grupo temático de contenido, que generalmente incluye una cabecera.*

Consideremos el siguiente marcado válido en HTML 4:

```
<h1>Rules for Munchkins</h1>
<h2>Yellow Brick Road</h2>
<p>It is vital that Dorothy follows it—so no selling
    bricks as "souvenirs"</p>
<h2>Fan Club uniforms</h2>
<p>All Munchkins are obliged to wear their "I’m a friend
    of Dorothy!" t-shirt when representing the club</p>
<p><strong>Vital caveat about the information above:
    does not apply on the first Thursday of the month.</strong></p>
```

En este caso, y desde un punto de vista semántico, es complicado deducir si el texto *Vital caveat about the information above: does not apply on the first Thursday of the month.* pertenece al contenido completo o está relacionado con la *sección Fan Club uniforms*. Gracias a la etiqueta `<section>`, es muy sencillo separar e identificar a qué sección pertenece cada contenido:

```
<article>
    <h1>Rules for Munchkins</h1>
    <section>
        <h2>Yellow Brick Road</h2>
        <p>It is vital that Dorothy follows it—so no selling
            bricks as "souvenirs"</p>
    </section>
    <section>
        <h2>Fan Club uniforms</h2>
        <p>All Munchkins are obliged to wear their "I’m a friend
            of Dorothy!" t-shirt when representing the club</p>
    </section>
    <p><strong>Vital caveat about the information above:
        does not apply on the first Thursday of the month.</strong></p>
</article>
```

```
<article>
    <h1>Rules for Munchkins</h1>
    <section>
        <h2>Yellow Brick Road</h2>
        <p>It is vital that Dorothy follows it—so no selling
            bricks as "souvenirs"</p>
    </section>
    <section>
        <h2>Fan Club uniforms</h2>
        <p>All Munchkins are obliged to wear their "I’m a friend
            of Dorothy!" t-shirt when representing the club</p>
        <p><strong>Vital caveat about the information above:
            does not apply on the first Thursday of the month.</strong></p>
    </section>
</article>
```

Como podemos observar en los dos ejemplos anteriores, es muy sencillo agrupar contenido que pertenece a una misma sección, permitiendo incluirlo dentro de un contexto semántico.

Otra de las posibilidades que nos ofrece esta etiqueta, es la de dividir nuestro documento en secciones, que incluyen contenido de temáticas diferentes entre sí. Si además queremos separar estos contenidos visualmente en dos columnas, lo lógico sería utilizar las tradicionales etiquetas `<div>` para agrupar los artículos según su temática, y posteriormente aplicar estilos CSS o JavaScript para presentarlos en forma de *pestañas*.

En este caso, la etiqueta `<div>` no nos aporta ningún significado semántico, tan sólo estructural. La etiqueta `<section>` es la encargada de añadir semántica en estos casos:

```
<section>
    <h1>Articles about llamas</h1>
    <article>
        <h2>The daily llama: Buddhism and South American camelids</h2>
        <p>blah blah</p>
    </article>
    <article>
        <h2>Shh! Do not alarm a llama</h2>
        <p>blah blah</p>
    </article>
</section>
<section>
    <h1>Articles about root vegetables</h1>
    <article>
        <h2>Carrots: the orange miracle</h2>
        <p>blah blah</p>
    </article>
```

```
    <article>
        <h2>Eat more Swedes (the vegetables, not the people)</h2>
        <p>blah blah</p>
    </article>
</section>
```

2.4.6 `<ASIDE>`

Según la especificación, un elemento `<aside>` representa lo siguiente:

> *Una sección de una página que consiste en contenido tangencialmente relacionado con el contenido alrededor del elemento, y puede considerarse separado de este contenido. Estas secciones son normalmente representadas como elementos laterales en medios impresos. Este elemento puede utilizarse contener citas, anuncios, grupos de elementos de navegación y cualquier otro contenido separado del contenido principal de la pagina.*

Dentro de un artículo, por ejemplo, puede ser utilizado para mostrar contenido relacionado como citas u otros artículos relacionados.

2.4.7 `<FIGURE>`

Según la especificación, un elemento `<figure>` representa lo siguiente:

> *The figure element represents some flow content, optionally with a caption, that is self-contained and is typically referenced as a single unit from the main flow of the document.*
>
> *The element can be used to annotate illustrations, diagrams, photos, code listings, etc. This includes, but is not restricted to, content referred to from the main part of the document, but that could, without affecting the flow of the document, be moved away from that primary content, e.g. to the side of the page, to dedicated pages, or to an appendix.*

Hasta ahora, no había una manera correcta de poder añadir un subtítulo o una leyenda a un contenido concreto, como explicar una figura o atribuir una imagen a un fotógrafo. Gracias a la etiqueta `<figure>` podemos contener una imagen (o un vídeo, ilustración o bloque de código) en un elemento y relacionarlo con un contenido concreto:

```
<figure>
    <img src="welcome.jpg" alt="Alternative text">
    <figcaption>
        Bruce and Remy welcome questions
            <small>Photo &copy; Bruce’s mum</small>
    </figcaption>
</figure>
```

En conveniente recordar que el atributo `alt` indica el texto a mostrar cuando la imagen no está disponible, y no está pensada para contener una descripción de la imagen, y mucho menos para duplicar lo ya indicado en la etiqueta `<figcaption>`.

Ejercicio 1

Ver enunciado

2.5 ATRIBUTOS GLOBALES

HTML5 también incluye nuevos atributos globales que pueden ser asignados a cualquier elemento. Son los siguientes:

2.5.1 ACCESSKEY

El atributo `accesskey` permite a los desarrolladores especificar un atajo de teclado que permite activar un elemento a asignarle el foco. Este atributo ya existía en HTML 4, aunque ha sido utilizado en muy pocas ocasiones. Como HTML5 está pensado para aplicaciones, y algunos usuarios siguen prefiriendo los atajos de teclado, este atributo no ha sido eliminado, y ahora está disponible para cualquier elemento.

Para evitar conflictos con otros atajos de teclado, o con los propios del navegador, ahora esta etiqueta permite asignar alternativas en este atributo. Un ejemplo incluido en la especificación:

```
<input type="search" name="q" accesskey="s 0">
```

Esto quiere decir que este elemento es accesible a través de dos atajos de teclado, a través de la tecla `s` o a través de la tecla `0` (en ese orden).

2.5.2 CONTENTEDITABLE

Inventado por Microsoft, e implementado por el resto de los navegadores, la etiqueta `contenteditable` es ahora parte de la especificación oficial.

La introducción de esta etiqueta significa principalmente dos cosas:

- Primero, los usuarios pueden editar los contenidos de un elemento que incluya esta etiqueta. Este elemento debe ser seleccionable y el navegador debe proporcionar una marca que indique la posición actual del cursor.
- Y segundo, es posible cambiar el formato del texto del contenido, añadiendo negritas, cambiar la fuente, añadir listas, etc.

Este atributo es de tipo *booleano*, por lo que su valor puede ser *true* o *false*. Al acceder desde JavaScript a este atributo, hay que tener en cuenta su notación *lowerCamelCase*, siendo el nombre de la propiedad del DOM `contentEditable`. Además, existe otra propiedad llamada `isContentEditable`, que indica si el elemento es editable o no.

Finalmente, el contenido que ha sido seleccionado por el usuario, puede ser objeto de modificaciones, como hemos comentado antes. A través del comando `element.execCommand()` es posible indicar el tipo de modificación (poner en negrita, copiar, cambiar la fuente...), siempre que el documento se haya indicado como editable.

```
document.designMode = 'on';
```

Si se desea almacenar los cambios realizados en el contenido, es necesario enviarlo al servidor. No existe ningún API o método en JavaScript que nos posibilite esta acción, por lo que debemos utilizar algún tipo de tecnología tipo AJAX.

2.5.3 DATA-* (CUSTOM DATA ATTRIBUTES)

HTML5 permite crear atributos personalizados para los elementos. Estos atributos son utilizados para pasar información a JavaScript. Como veremos en el capítulo correspondiente, hasta ahora se utilizaba el atributo *class* para de alguna manera almacenar información asociada con elementos, pero esto cambia radicalmente con estos atributos.

```
<ul id="vegetable-seeds">
  <li data-spacing="10cm" data-sowing-time="March to June">Carrots</li>
  <li data-spacing="30cm" data-sowing-time="February to March">Celery</li>
  <li data-spacing="3cm" data-sowing-time="March to September">Radishes</li>
</ul>
```

2.5.4 DRAGGABLE

Este atributo indica que el elemento indicado puede ser *arrastable*. Lo veremos en el capítulo correspondiente.

ÍNDICE DE CONTENIDOS

CAPÍTULO 3 ELEMENTOS DE FORMULARIO

HTML5 hace que el desarrollo de formularios sea mucho más sencillo. Se han añadido dos nuevos métodos que pueden ser utilizados en la acción del formulario (`update` y `delete`), pero lo más interesante son los nuevos tipos de `input` y elementos de formulario que mejoran la experiencia del usuario y facilitan el desarrollo de los formularios. Estos nuevos elementos añaden en algunos casos, validaciones propias de sus datos, por lo que ya no es necesario JavaScript para realizar este proceso.

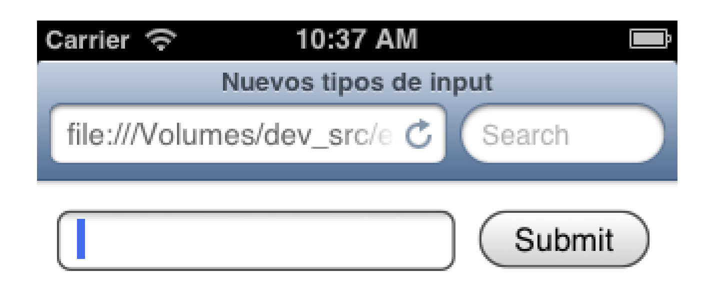
3.1 NUEVOS TIPOS DE INPUT

La especificación de HTML5 define 12 nuevos tipos de `input` que podemos utilizar en nuestros formularios. Esta especificación no define cómo deben mostrarse los nuevos tipos en los navegadores, ni los campos ni las validaciones. De hecho, y gracias a cómo está especificado HTML, los navegadores que no *comprendan* los nuevos tipos de entrada, mostrarán un campo de texto tradicional, por lo que la compatibilidad con navegadores antiguos está garantizada.

3.1.1 TIPO `EMAIL`

El nuevo tipo `<input type="email">` indica al navegador que no debe permitir que se envíe el formulario si el usuario no ha introducido una dirección de email *válida*, pero no comprueba si la dirección existe o no, sólo si el formato es válido. Como ocurre con el resto de campos de entrada, puede enviar este campo vacío a menos que se indique que es obligatorio.

El atributo `multiple` indica que el valor de este campo, puede ser una lista de emails válidos, separados por comas.



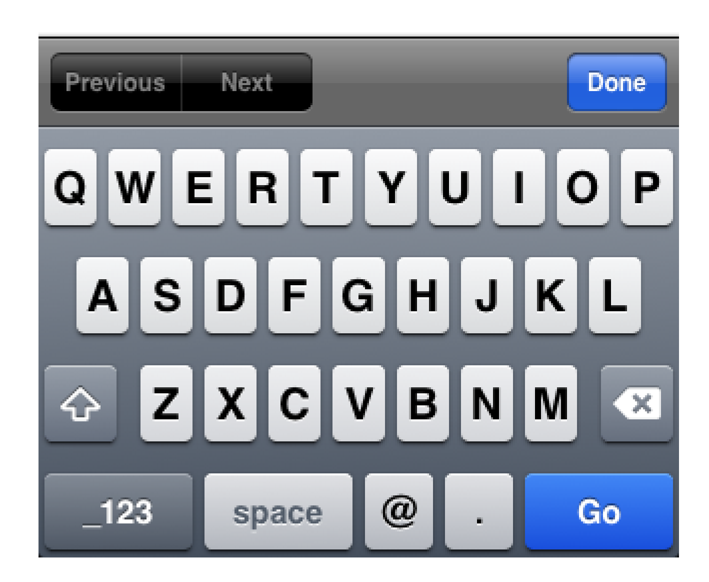
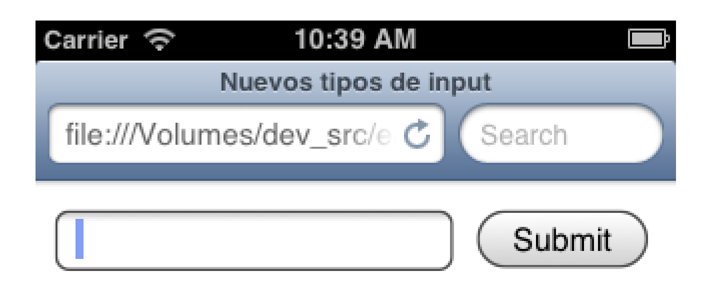


Figura 3.1 Campo de tipo `email` en un dispositivo iOS

3.1.2 TIPO `URL`

El nuevo tipo `<input type="url">` indica al navegador que no debe permitir que se envíe el formulario si el usuario no ha introducido una URL correcta. Algunos navegadores ofrecen ayudas al usuario, como Opera que añade el prefijo `http://` a la URL si el usuario no lo ha introducido. Una URL no tiene que ser necesariamente una dirección web, sino que es posible utilizar cualquier formato de URI válido, como por ejemplo `tel:555123456`.



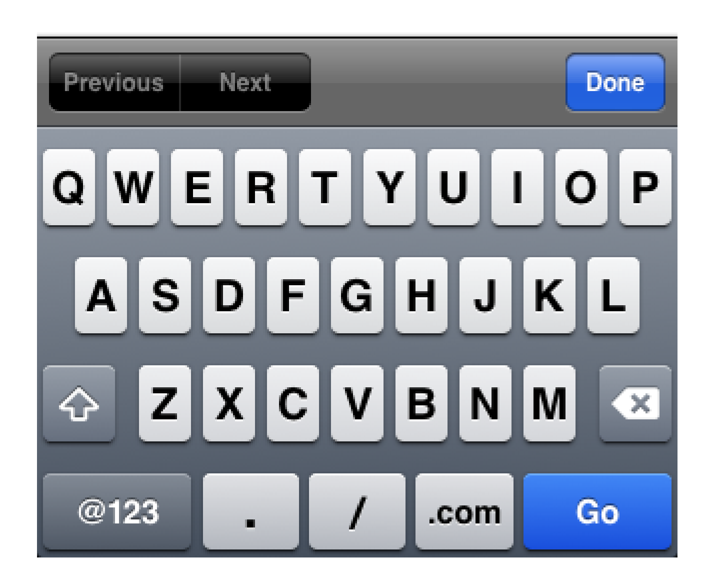
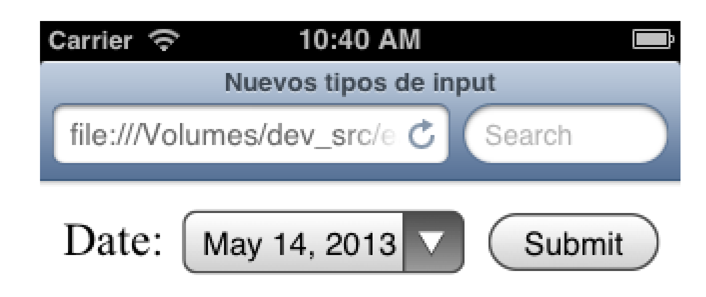


Figura 3.2 Campo de tipo `url` en un dispositivo iOS

3.1.3 TIPO `DATE`

El nuevo tipo `<input type="date">` es de los más esperados y útiles. En muchos de los sitios web es normal disponer de campos específicos de fecha, donde el usuario debe especificar fechas (para un concierto, vuelo, reserva de hotel, etc). Al existir tantos formatos de fecha diferentes (DD-MM-YYYY o MM-DD-YYYY o YYYY-MM-DD), esto puede suponer un inconveniente para los desarrolladores o los propios usuarios.

Este nuevo tipo de campo resuelve estos problemas, ya que es el navegador el que proporciona la interfaz de usuario para el calendario, e independientemente del formato en el que se muestre, los datos que se envían al servidor cumplen la norma ISO para el formato de fechas.



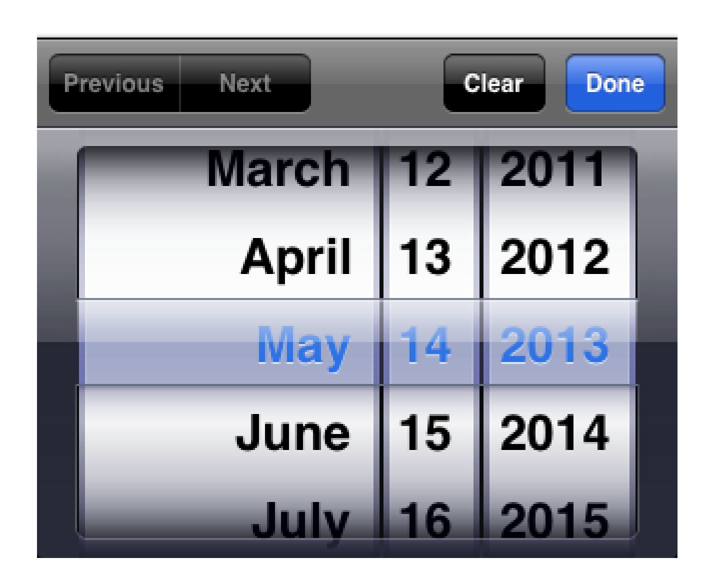


Figura 3.3 Campo de tipo `date` en un dispositivo iOS

3.1.4 TIPO `TIME`

El nuevo tipo `<input type="time">` permite introducir una hora en formato 24h, y validarlo. De nuevo, es el navegador el encargado de mostrar la interfaz de usuario correspondiente: puede ser un simple campo donde es posible introducir la hora y los minutos, o mostrar algo más complejo como un reloj de agujas.

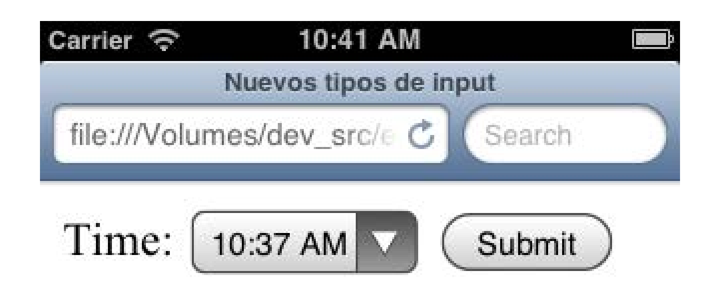
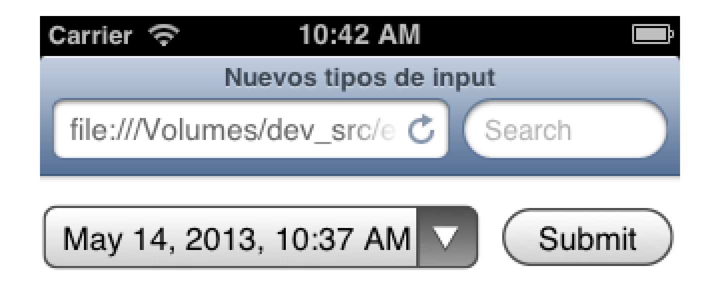




Figura 3.4 Campo de tipo `time` en un dispositivo iOS

3.1.5 TIPO `DATETIME`

Este nuevo tipo de campo es la combinación de los tipos `date` y `time`, por lo que se valida tanto la fecha como la hora introducida.



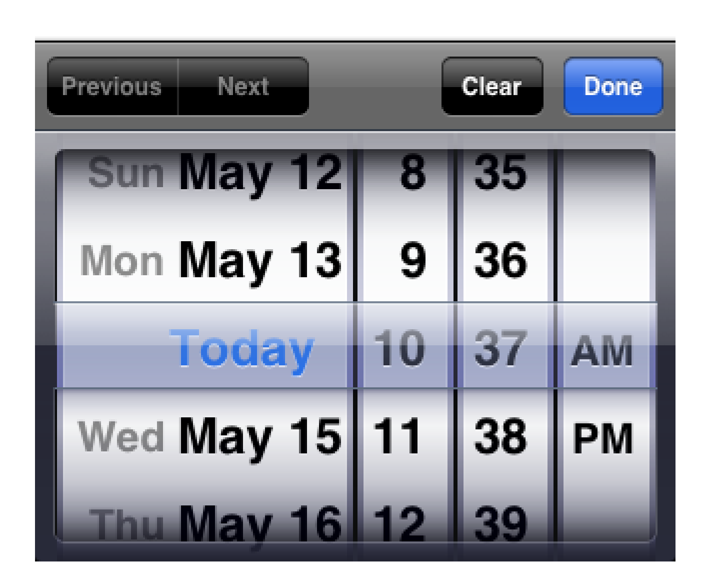
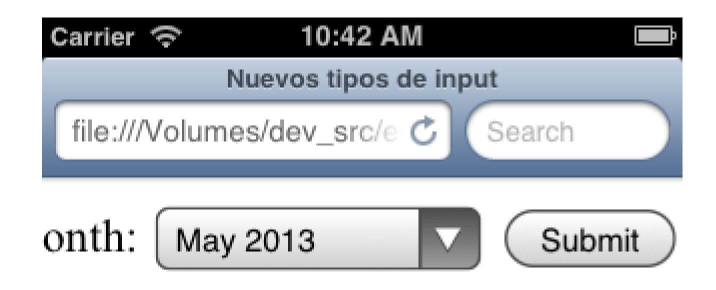


Figura 3.5 Campo de tipo `datetime` en un dispositivo iOS

3.1.6 TIPO `MONTH`

El nuevo tipo `<input type="month">` permite la selección de un mes en concreto. La representación interna del mes es un valor entre 1 y 12, pero de nuevo queda en manos del navegador la manera de mostrarlo al usuario, utilizando los nombres de los meses por ejemplo.



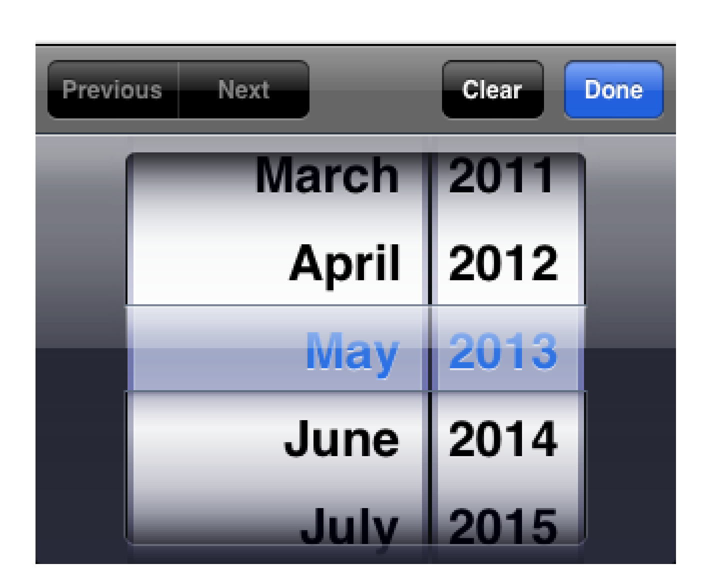


Figura 3.6 Campo de tipo `month` en un dispositivo iOS

3.1.7 TIPO `WEEK`

El nuevo tipo `<input type="week">` permite la selección de una semana del año concreta. La representación interna del mes es un valor entre 1 y 53, pero de nuevo queda en manos del navegador la manera de mostrarlo al usuario. La representación interna de la semana 7, por ejemplo, es la siguiente: `2013-W07`.

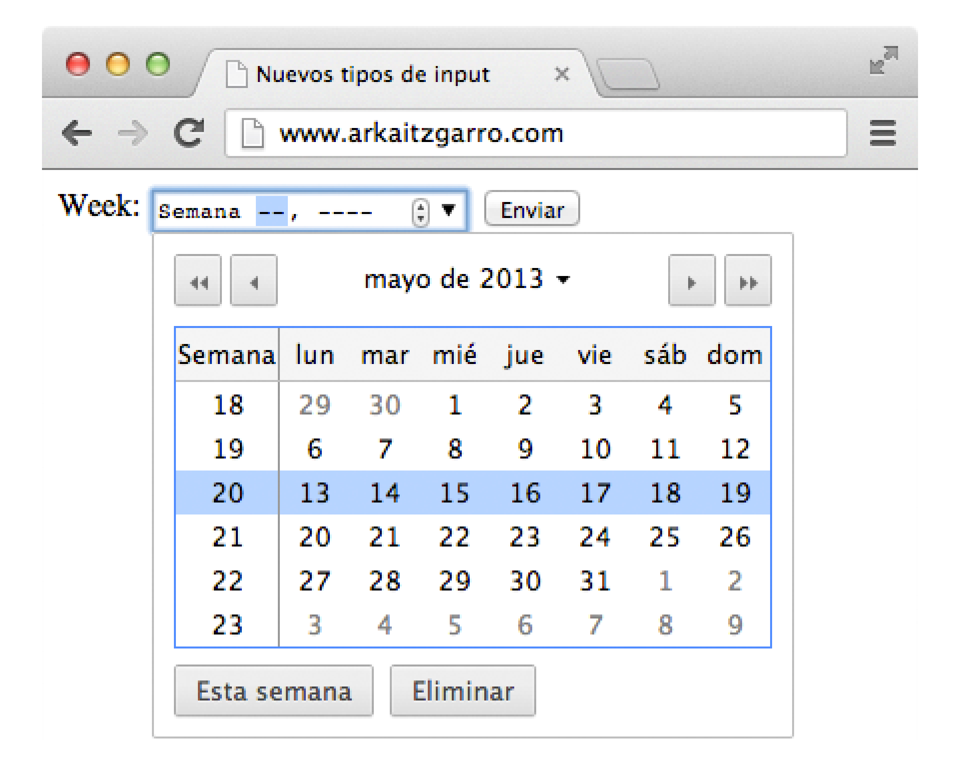


Figura 3.7 Campo de tipo `week` en Google Chrome

3.1.8 TIPO `NUMBER`

Como es de esperar, el nuevo tipo `<input type="number">` valida la entrada de un tipo de dato numérico. Este tipo de campo encaja perfectamente con los atributos `min`, `max` y `step`, que veremos más adelante.





Figura 3.8 Campo de tipo `number` en un dispositivo iOS

3.1.9 TIPO `RANGE`

El nuevo tipo `<input type="range">`, muestra un control deslizante en el navegador. Para conseguir un elemento de este estilo, era necesario un gran esfuerzo para combinar imágenes, funcionalidad y accesibilidad, siendo ahora mucho más sencillo. Este tipo de campo encaja de nuevo perfectamente con los atributos `min`, `max` y `step`, que veremos más adelante.

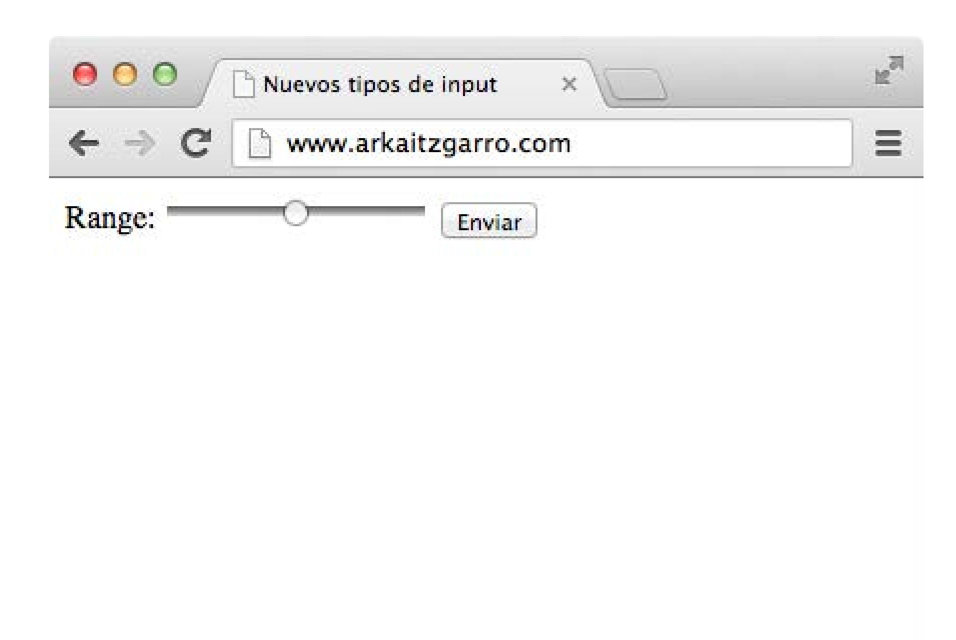


Figura 3.9 Campo de tipo `range` en Google Chrome

3.1.10 TIPO `TEL`

El nuevo tipo `<input type="tel">`, espera que se proporcione un número de teléfono. No se realiza ninguna validación, ni se obliga a que únicamente se proporcionen caracteres numéricos, ya que un número de teléfono puede representarse de muchas maneras: *+44 (0) 208 123 1234*.

La gran ventaja de este campo es que en dispositivos con un teclado virtual, éste se adaptará para mostrar únicamente los caracteres asociados on números de teléfono.





Figura 3.10 Campo de tipo `tel` en un dispositivo iOS

3.1.11 TIPO `SEARCH`

El nuevo tipo `<input type="search">`, espera que se proporcione un término de búsqueda. La diferencia con un campo de texto normal, es únicamente estética, aunque puede ofrecer alguna funcionalidad extra como un histórico de últimos términos introducidos o una ayuda para el borrado. Por norma general, toma el aspecto de un campo de búsqueda del navegador o sistema operativo.

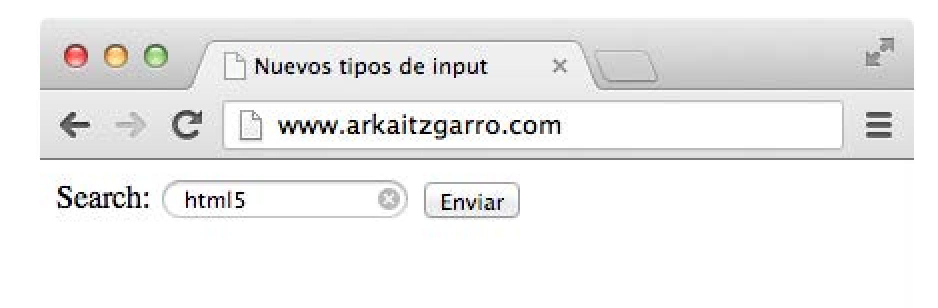


Figura 3.11 Campo de tipo `search` en Google Chrome

3.1.12 TIPO `COLOR`

El nuevo tipo `<input type="color">`, permite seleccionar un color de una paleta de colores mostrada por el navegador. Esta paleta de colores, coincide, por norma general, con la interfaz de selección de colores del sistema operativo.

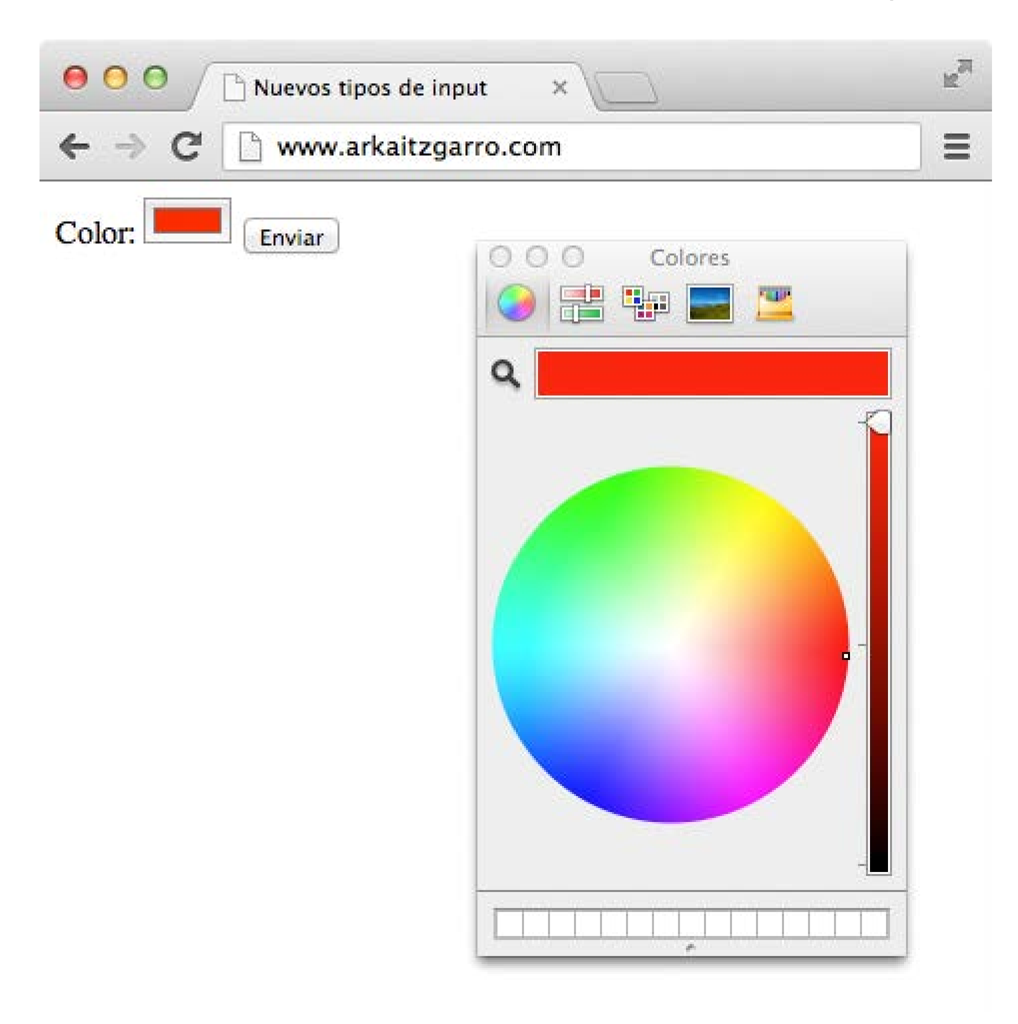


Figura 3.12 Campo de tipo `color` en Google Chrome

3.2 NUEVOS ATRIBUTOS

Al igual que los nuevos tipos de campo, el elemento `input` ha recibido nuevos atributos para definir su comportamiento y restricciones: `autocomplete`, `min`, `max`, `multiple`, `pattern`, `autofocus`, `placeholder`, `required` y `step`. Existe además un nuevo atributo, `list`, que hace referencia a otro elemento, permitiendo crear un nuevo tipo de entrada de datos.

3.2.1 ATRIBUTO `LIST` Y `<DATALIST>`

La combinación del atributo `list` y un elemento de tipo `<datalist>` da como resultado un campo de texto, donde el usuario puede introducir cualquier contenido, y las opciones definidas en el `<datalist>` se muestran como una lista desplegable. Hay que tener en cuenta que la lista tiene que estar contenida en un elemento `<datalist>` cuyo

`id` coincide con el indicado en el atributo `list`:

```
<input id="form-person-title" type="text" list="mylist">
<datalist id="mylist">
    <option label="Mr" value="Mr">
    <option label="Ms" value="Ms">
    <option label="Prof" value="Mad Professor">
</datalist>
```

En este ejemplo se utiliza un campo de tipo `text`, pero puede ser utilizado igualmente con campos de tipo `url` y `email`.

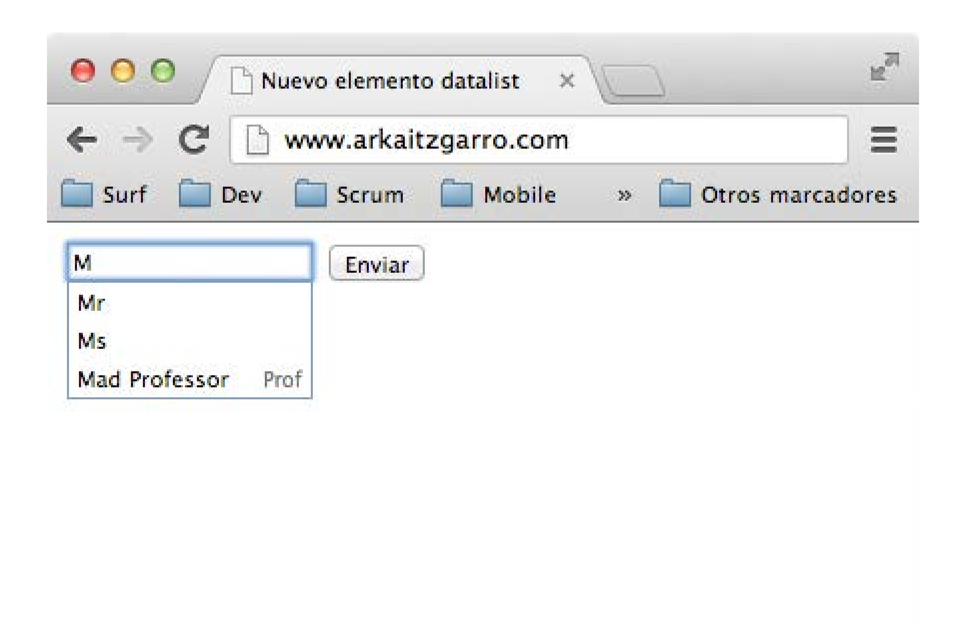


Figura 3.13 Nuevo elemento `datalist`, asociado a un campo de texto

3.2.2 ATRIBUTO `AUTOFOCUS`

El atributo *booleano* `autofocus` permite definir que control va a tener el foco cuando la página se haya cargado. Hasta ahora, esto se conseguía a través de JavaScript, utilizando el método `.focus()` en un elemento concreto, al cargarse el documento. Ahora es el navegador el encargado de esta tarea, y puede comportarse de manera más inteligente, como no cambiando el foco de un elemento si el usuario ya se encuentra escribiendo en otro campo (éste era un problema común con JavaScript).

Únicamente debe existir un elemento con este atributo definido en el documento. Desde el punto de vista de la usabilidad, hay que utilizar este atributo con cuidado. Hay que utilizarlo únicamente cuando el control que recibe el foco es el elemento principal de la página, como en un buscador, por ejemplo.

3.2.3 ATRIBUTO `PLACEHOLDER`

Una pequeña mejora en la usabilidad de los formularios, suele ser colocar un pequeño texto de ayuda en algunos campos, de manera *discreta* y que desaparece cuando el

usuario introduce algún dato. Como con el resto de elementos, hasta ahora era necesario utilizar JavaScript para realizar esta tarea, pero el atributo `placeholder` resuelve esta tarea.

Es importante recordar que este atributo no sustituye a la etiqueta `<label>`.

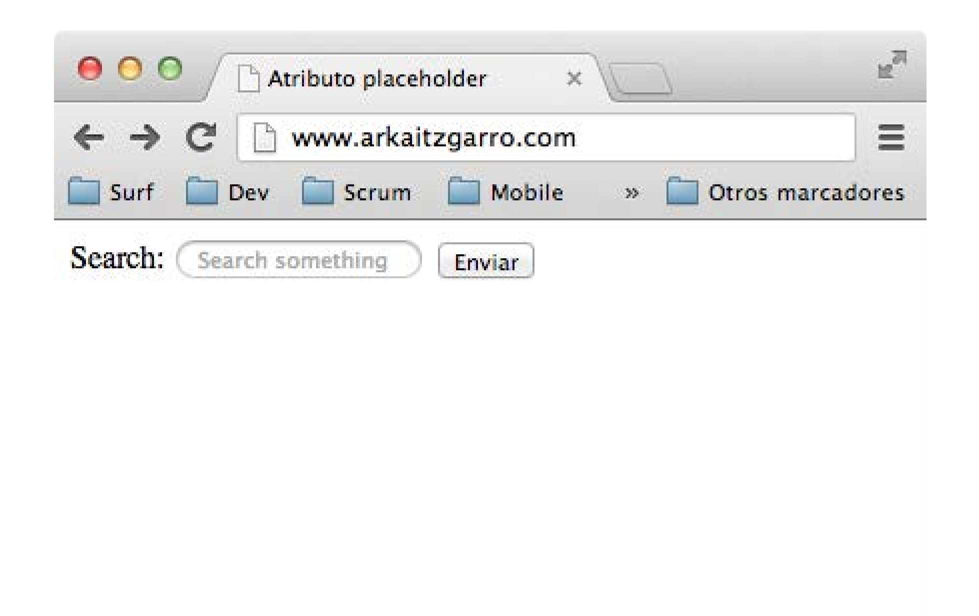


Figura 3.14 Atributo `placeholder` en un campo de búsqueda

3.2.4 ATRIBUTO `REQUIRED`

Este atributo puede ser utilizado en un `<textarea>` y en la gran mayoría de los elementos `<input>` (excepto en los de tipo `hidden`, `image` o botones como `submit`). Cuando este atributo está presente, el navegador no permite el envío del formulario si el campo en concreto está vacío.

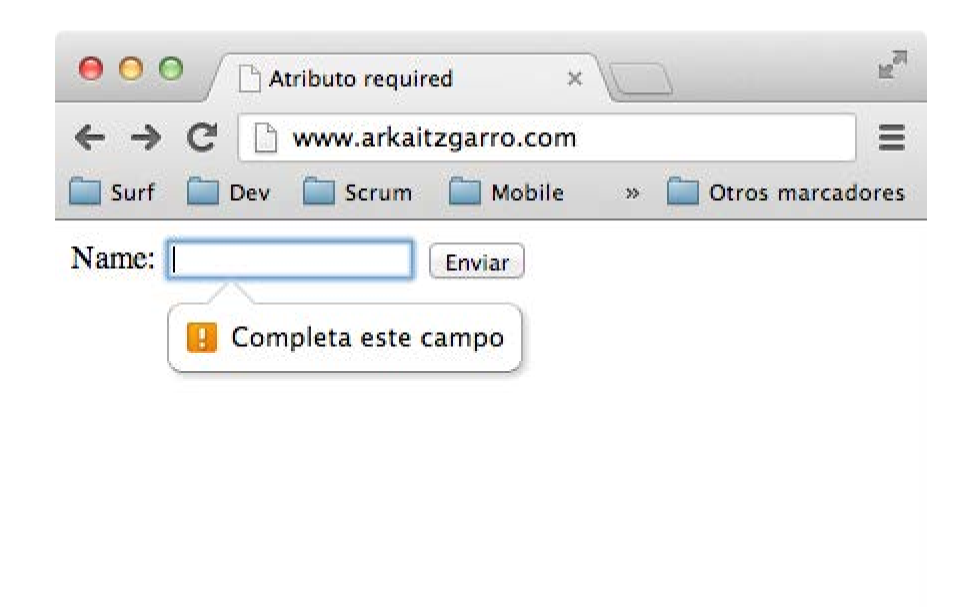


Figura 3.15 Atributo `required` en un campo de texto

3.2.5 ATRIBUTO `MULTIPLE`

Este atributo permite definir que un campo puede admitir varios valores, como URLs o emails. Un uso muy interesante de este atributo es utilizarlo en conjunto con el campo `<input type="file">`, ya que de esta manera nos permite seleccionar varios ficheros que podemos enviar al servidor al mismo tiempo.

```
<input type="file" multiple="multiple">
```

3.2.6 ATRIBUTO `AUTOCOMPLETE`

Algunos navegadores suelen incluir alguna funcionalidad de autocompletado en algunos campos de formulario. A pesar de haber sido introducido recientemente en el estándar de HTML5, es una característica que lleva mucho tiempo siendo utilizada, concretamente desde la versión 5 de Internet Explorer.

Este atributo permite controlar el comportamiento del autocompletado en los campos de texto del formulario (que por defecto está activado).

3.2.7 ATRIBUTOS `MIN` Y `MAX`

Como hemos visto en el campo `<input type="number">`, estos atributos restringen los valores que pueden ser introducidos; no es posible enviar el formulario con un valor menor que `min` o un valor mayor que `max`. También es posible utilizarlo en otro tipo de campos como `date`, para especificar fechas mínimas o máximas.

```
<input type="date" min="2010-01-01" max="2010-12-31">
```



Figura 3.16 Atributos `min` y `max` en un campo numérico

3.2.8 ATRIBUTO `STEP`

El atributo `step` controla los *pasos* por los que un campo aumenta o disminuye su valor. Si un usuario quiere introducir un porcentaje, pero queremos que sea en múltiplos de 5, lo haríamos de la siguiente manera:

```
<input type="range" mix="0" max="100" step="5">
```

3.2.9 ATRIBUTO `PATTERN`

Algunos de los tipos de `input` que hemos visto anteriormente (`email`, `number`, `url`...), son realmente expresiones regulares que el navegador evalúa cuando se introducen datos. El atributo `pattern` nos permite definir una expresión regular que el valor del campo debe cumplir. Por ejemplo, si el usuario debe introducir un número seguido de tres letras mayúsculas, podríamos definir esta expresión regular:

```
<input pattern="[0-9][A-Z]{3}" name="part"
    title="A part number is a digit followed by three uppercase letters.">
```

La especificación indica que la sintaxis de la expresión regular que utilicemos debe coincidir con la utilizada en JavaScript.

3.2.10 ATRIBUTO `FORM`

Tradicionalmente, los campos de un formulario van incluidos dentro de la correspondiente etiqueta `<form>`. Si por la razón que fuese (principalmente diseño) un elemento tuviese que mostrarse apartado del resto de elementos del formulario, se hacía casi necesario incluir toda la página dentro de una etiqueta `<form>`.

Con HTML5, los elementos que hasta ahora era obligatorio que estuviesen contenidos dentro del elemento `<form>`, pueden colocarse en cualquier lugar de la página, siempre que los relacionemos con el formulario concreto a través del atributo `form` y el `id` de dicho formulario.

```
<form id="foo">
    <input type="text">
    ...
</form>
<textarea form="foo"></textarea>
```

En este caso, el elemento `<textarea>` se encuentra fuera del formulario, pero realmente pertenece a él ya que hemos definido el atributo `form` con el identificador del formulario.

3.3 NUEVOS ELEMENTOS

Además de los nuevos tipos de `input` incluidos en la especificación, se han añadido nuevos elementos de formulario, entre los que se incluyen los siguientes:

3.3.1 <PROGRESS>

El elemento `<progress>` es utilizado para representar un *avance* o *progreso* en la ejecución de una tarea, como puede ser la descarga de un fichero o la ejecución de una tarea compleja. Define los siguientes atributos:

- `max`: define el trabajo total a realizar por la tarea, la duración de un vídeo... Su valor por defecto es `1.0`.
- `value`: el valor actual (en coma flotante) o estado del progreso. Su valor debe ser mayor o igual a `0.0` y menor o igual a `1.0` o el valor especificado en `max`.
- `position`: atributo de sólo lectura que representa la posición actual del elemento `<progress>`. Este valor es igual a `value`/`max` y `-1` si no se puede determinar la posición.

Las unidades son arbitrarias, y no se especifican.

```
<progress value="5" max="20">5</progress>
```

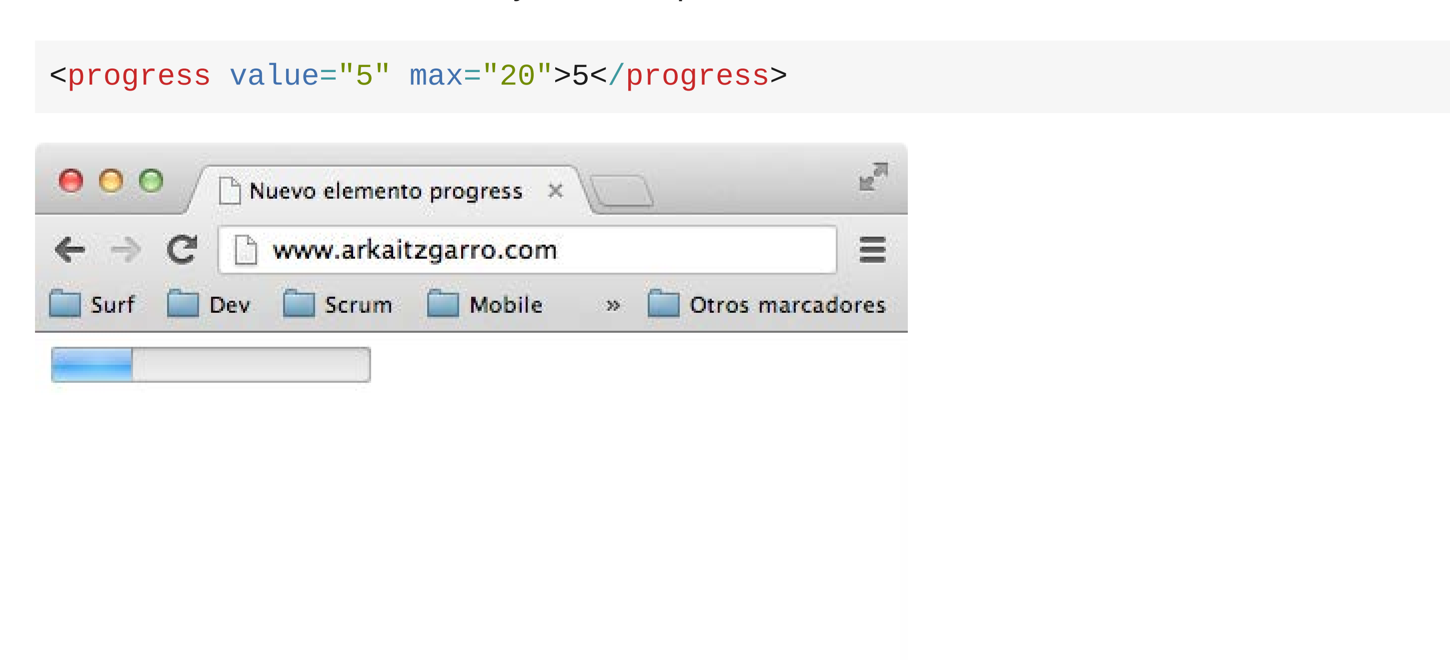


Figura 3.17 Nuevo elemento progress

3.3.2 <METER>

El elemento `<meter>` es muy similar a `<progress>` (de hecho, se discute la necesidad de disponer de dos elementos tan similares). Esta nueva etiqueta se usa para representar escalas de medidas conocidas, como la longitud, masa, peso, uso de disco, entre otras. Define los siguientes atributos:

- `value`: representa el valor actual. Si no se especifica, se toma como valor el primer número que aparece en el contenido del elemento. Su valor por defecto es `0`.
- `min`: el mínimo valor permitido. El valor por defecto es `0`.

- `max`: el mayor valor permitido. Si no se especifica, su valor por defecto es `1`, a menos que el valor mínimo definido sea mayor que `1`, en cuyo caso el valor de `max` será igual a `min`.
- `low`: es considerado el límite inferior del rango de valores.
- `max`: es considerado el límite superior del rango de valores.
- `optimum`: representa el valor óptimo del elemento, y se encuentra entre `min` y `max`.

```
<p>Your score is:
    <meter value="91" min="0" max="100"
            low="40" high="90" optimum="100">A+</meter>
</p>
```

Figura 3.18 Nuevo elemento meter

Ejercicio 2

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 4 QUÉ ES MODERNIZR

Modernizr es una librería JavaScript que nos permite conocer la compatibilidad del navegador con tecnologías **HTML5** y **CSS3**, lo que nos permitirá desarrollar sitios web que se adapten a las capacidades cada navegador.

Este *framework* es un paquete de detección de las capacidades de un navegador relativas a HTML5 y CSS3, esto es, una librería JavaScript que nos informará cuáles de las funcionalidades de estas tecnologías están disponibles en el navegador del usuario, para utilizarlas, o no, en cada caso.

Sabiendo que nuestro navegador soporta ciertas capacidades de CSS3 o de HTML5, podremos utilizarlas con libertad. De modo contrario, si sabemos que un navegador no es compatible con determinada funcionalidad, podremos implementar variantes que sí soporte y así crear sitios web que se adaptan perfectamente al cliente web de cada visitante.

Existen dos herramientas principales en Modernizr que se pueden utilizar para detectar las funcionalidades que están presentes en un navegador. Una la podemos utilizar a través de JavaScript y otra directamente sobre código CSS. En resumen, con Modernizr podemos detectar las funcionalidades disponibles de CSS3 y HTML5.

4.1 AÑADIR MODERNIZR A UNA PÁGINA

El primer paso consistirá en descargar el archivo con el código fuente de Modernizr. Se trata de un archivo con código JavaScript que podemos encontrar en dos variantes:

- **Development**: contiene el código fuente completo, sin comprimir y con comentarios. Debemos utilizar esta variante únicamente en desarrollo o cuando queremos acceder a su código, para comprenderlo o ampliarlo.
- **Production**: es recomendable (o más bien obligatorio) utilizar esta variante cuando pasamos a un entornos de producción. Al descargarnos Modernizr, tenemos la posibilidad de generar una librería únicamente con las funcionalidades que queremos detectar, lo que nos permitirá ahorrarnos una importante cantidad de `KB` innecesarios.

Una vez que hemos descargado nuestra librería, debemos incluirla en el código HTML de la página, de la misma manera que incluimos scripts JavaScript.

```
<script src="/js/lib/vendor/modernizr-custom.min.js"></script>
```

> *Según podemos leer en la documentación de Modernizr, se aconseja colocar el script dentro del HEAD, porque debe cargarse antes del BODY de la página, debido a un componente que quizás utilicemos, para permitir HTML5 en Internet Explorer, llamado HTML5 Shiv. Además, se recomienda colocarlo después de los estilos CSS para evitar un comportamiento poco deseable llamado FOUC, por el cual puede mostrarse, por un pequeño espacio de tiempo, la página sin los estilos CSS aplicados.*

A partir de este momento tendremos disponibles nuestros scripts de detección de funcionalidades así como una serie de clases CSS que nos ayudarán a aplicar estilos solo cuando los navegadores los soporten.

4.2 OBJETO MODERNIZR

Cuando tenemos Modernizr cargado en nuestra página, se crea automáticamente un objeto JavaScript que tiene una serie de propiedades que nos indican si están o no disponibles cada una de las funcionalidades presentes en CSS3 y HTML5. Las mencionadas propiedades contienen simplemente valores booleanos (*true* o *false*) que podemos consultar para saber si están o no disponibles las funcionalidades que deseamos utilizar.

El uso es tan sencillo como se indica a continuación:

```
if (Modernizr.boxshadow) {
   // Podemos aplicar sombras!
} else {
   // La propiedad box-shadow no está disponible
}
```

Aquí estamos consultando la propiedad `boxshadow` del objeto Modernizr. Esta propiedad nos indica si el navegador es compatible con el atributo `box-shadow` de CSS3, que sirve para crear cajas de contenido con sombreado.

Ahora veamos otro ejemplo similar que detectaría si está disponible el elemento `canvas` del HTML5.

```
if (Modernizr.canvas) {
   // Podemos utilizar canvas!
} else {
   // El elemento canvas no está disponible
}
```

Este es un simple ejemplo que comprueba si están disponibles ciertas propiedades y funcionalidades de CSS3 y HTML5. Lógicamente, tendremos que desarrollar las funcionalidades que correspondan en cada caso. El listado completo de propiedades del

objeto para la detección de funcionalidades HTML5 y CSS3 se puede encontrar en la propia documentación de Modernizr.

4.3 CLASES CSS EN MODERNIZR

Cuando tenemos Modernizr cargado en nuestra página, éste crea automáticamente una serie de clases que asigna al elemento `html` del documento. Cada una de estas clases hace referencia a las características que soporta el navegador, permitiendo desde CSS adaptar la interfaz según las funcionalidades. Un ejemplo de las clases que crea en un navegador de escritorio moderno:

```
<html lang="en" class=" js flexbox canvas canvastext webgl no-touch geolocation
 postmessage websqldatabase indexeddb hashchange history draganddrop websockets
 rgba hsla multiplebgs backgroundsize borderimage borderradius boxshadow textsh
adow opacity cssanimations csscolumns cssgradients cssreflections csstransforms
 csstransforms3d csstransitions fontface generatedcontent video audio localstor
age sessionstorage webworkers applicationcache svg inlinesvg smil svgclippaths"
>
<head>
    ...
    <script src="/js/lib/vendor/modernizr-custom.min.js"></script>
</head>
<body>
    <div class="elemento"></div>
</body>
</html>
```

Todo el proceso es automático y la creación de cada una de esas clases se realizará únicamente en caso de que el navegador sea compatible con cada una de las características de CSS3 y HTML5.

La manera de utilizar estas clases es muy sencilla. Pensemos en que queremos aplicar sombra a un elemento con CSS3 en los navegadores que lo permitan y emular ese sombreado por medio de estilos CSS clásicos en los navegadores que no soporten el atributo `box-shadow`. Lo único que tenemos que hacer es aplicar los estilos *clásicos* al elemento que deseemos:

```
.elemento{
   border-left: 1px solid #ccc;
   border-top: 1px solid #ccc;
   border-bottom: 1px solid #666;
   border-right: 1px solid #666;
}
```

Posteriormente, si nuestro navegador es compatible con el atributo `box-shadow` de CSS3, Modernizr habrá incluido la clase "boxshadow" en el elemento `html`. Nosotros podemos

utilizar dicha clase CSS para aplicar estilos que sabemos que solo acabarán afectando a los navegadores que soporten el atributo `box-shadow`.

```
.boxshadow .elemento{
   border: 1px solid #ccc;
   box-shadow: #999 3px 3px 3px;
}
```

Como se puede ver, hemos sobrescrito la regla CSS para el borde y además hemos aplicado la propiedad `box-shadow`. El efecto conseguido es que los navegadores modernos, que son capaces de procesar el atributo `box-shadow`, mostrarán una sombra CSS3 y los no compatibles con esta propiedad al menos mostrarán unos estilos para los que sí son compatibles.

Ejercicio 3

Ver enunciado

4.4 EL MÉTODO `LOAD()`

El método `Modernizr.load()` es una sencilla manera para cargar librerías sólo cuando los usuarios las necesitan, es decir, cuando una funcionalidad en concreto está (o no) disponible. Es una buena manera de ahorrar ancho de banda y mejorar un poco más el rendimiento de la aplicación.

Por ejemplo, pensemos en que estamos desarrollando una aplicación basada en el API de geolocalización de HTML5. Con Modernizr podemos saber si el navegador ofrece soporte a ese API, mediante la propiedad `Modernizr.geolocation`. Si deseamos cargar unos recursos u otros dependiendo de la disponibilidad de esta funcionalidad, el método `Modernizr.load()` nos podrá ahorrar algo de código fuente y de paso acelerar nuestra página en algunas ocasiones. Con éste método podemos indicar a los navegadores que no soporten ese API que carguen el *polyfill* correspondiente, de modo que se pueda utilizar esa característica de HTML5 en ellos también.

La sintaxis de `Modernizr.load()` es bastante sencilla de comprender. Un ejemplo sencillo:

```
Modernizr.load({
  test: Modernizr.geolocation,
  yep : 'geo.js',
  nope: 'geo-polyfill.js'
});
```

En este ejemplo, se decide que *script* se debe cargar, en función de si la geolocalización esta soportada por el navegador o no. De esta manera, nos ahorramos el tener que

descargar código que el navegador no soporta y por lo tanto, no necesita.

`Modernizr.load()` es simple y eficaz, pero podemos utilizarlo de manera más compleja, como se muestra en el siguiente ejemplo:

```
// Give Modernizr.load a string, an object, or an array of strings and objects
Modernizr.load([
  // Presentational polyfills
  {
    // Logical list of things we would normally need
    test : Modernizr.fontface && Modernizr.canvas && Modernizr.cssgradients,
    // Modernizr.load loads css and javascript by default
    nope : ['presentational-polyfill.js', 'presentational.css']
  },
  // Functional polyfills
  {
    // This just has to be truthy
    test : Modernizr.websockets && window.JSON,
    // socket-io.js and json2.js
    nope : 'functional-polyfills.js',
    // You can also give arrays of resources to load.
    both : [ 'app.js', 'extra.js' ],
    complete : function () {
      // Run this after everything in this group has downloaded
      // and executed, as well everything in all previous groups
      myApp.init();
    }
  },
  // Run your analytics after you've already kicked off all the rest
  // of your app.
  'post-analytics.js'
]);
```

Ejercicio 4

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 5 DATASET

Gracias a HTML5, ahora tenemos la posibilidad de incorporar atributos de datos personalizados en todos los elementos HTML. Hasta la aparición de estos atributos, la manera de lograr un comportamiento similar (asociar datos a elementos), era incluir estos datos como clases CSS en los elementos, y acceder a ellos a través de jQuery, de una manera como la siguiente:

```
<input class="spaceship shields-5 lives-3 energy-75">
```

Una vez definidos los "atributos", era necesario acceder a estas clases y realizar un trabajo extra para extraer su nombre y su valor (convertir `energy-75` en `energy = 75`).

Afortunadamente, esto ya no es necesario, gracias a los atributos `dataset`. Estos nuevos atributos de datos personalizados constan de dos partes:

- Nombre del atributo: el nombre del atributo de datos debe ser de al menos un carácter de largo y debe tener el prefijo `data-`. No debe contener letras mayúsculas.
- Valor del atributo: el valor del atributo puede ser cualquier *string* o cadena. Con esta sintaxis, podemos añadir cualquier dato que necesitemos a nuestra aplicación, como se muestra a continuación:

```
<ul id="vegetable-seeds">
  <li data-spacing="10cm" data-sowing-time="March to June">Carrots</li>
  <li data-spacing="30cm" data-sowing-time="February to March">Celery</li>
  <li data-spacing="3cm" data-sowing-time="March to September">Radishes</li>
</ul>
```

Ahora podemos usar estos datos almacenados en nuestro sitio para crear una experiencia de usuario más rica y atractiva. Imagina que cuando un usuario hace clic en un "vegetable", una nueva capa se abre en el explorador que muestra la separación de semillas e instrucciones de siembra. Gracias a los atributos `data-` que hemos añadido a nuestros elementos `<li>`, ahora podemos mostrar esta información al instante sin tener que preocuparnos de hacer ninguna llamada `AJAX` y sin tener que hacer ninguna consulta a las bases de datos del servidor.

Prefijar los atributos personalizados con `data-` asegura que van a ser completamente ignorados por el agente de usuario. Por lo que al navegador y al usuario final de la web se refiere, no existe esta información.

> *Custom data attributes are intended to store custom data private to the page or application, for which there are no more appropriate attributes or elements. These attributes are not intended for use by software that is independent of the site that uses the attributes. Every HTML element may have any number of custom data attributes specified, with any value.*

W3C Specification

Esto es, los atributos de datos personalizados están destinadas a almacenar los datos personalizados que son de interés exclusivamente para la página o la aplicación, para los que no hay atributos o elementos más apropiados. Estos atributos no están destinados para un uso externo (a través de un software independiente al sitio que los utiliza). Cada elemento HTML puede tener un número indefinido de atributos de datos personalizados, con cualquier valor.

5.1 UTILIZACIÓN DE LOS *DATA ATTRIBUTES*

Como los atributos de datos personalizados son válidos en HTML5, pueden ser utilizados en cualquier navegador que soporte *HTML5 doctypes*. Estas son algunas de las formas en las que pueden ser utilizados:

- Para almacenar la altura inicial o la opacidad de un elemento que pudiera ser necesaria en los cálculos de animación JavaScript posteriores.
- Para almacenar los parámetros para una película de Flash que se carga a través de JavaScript.
- Para almacenar los datos estadísticos de una web.
- Para almacenar los datos acerca de la salud, munición o vida de un elemento en un juego JavaScript.
- Para poder añadir subtítulos a un `<video>`.

Y éstas algunas de las situaciones para las que no se deben usar los *data attributes*:

- Los atributos de datos personalizados no deben usarse si hay un atributo o elemento existente que es más adecuado.
- Éstos tampoco tienen la intención de competir con microformatos. En la especificación queda claro que los datos no están pensados para ser usados públicamente. El software externo no debe interactuar con ellos.
- La presencia/ausencia de un atributo de datos personalizado no se deben utilizar como una referencia para los estilos de CSS. Si se hace, podría sugerir que los datos que se están almacenando son de importancia inmediata para el usuario y se deberían marcar de una manera más accesible.

5.2 *DATA ATTRIBUTES* Y JAVASCRIPT

Ahora que comprendemos el funcionamiento de los atributos de datos personalizados y cuándo se utilizan, deberíamos centrarnos en cómo interactuar con ellos utilizando JavaScript.

Si quisiéramos recuperar o actualizar estos atributos utilizando JavaScript, podríamos hacerlo utilizando los métodos `getAttribute` y `setAttribute`.

```
<div id="strawberry-plant" data-fruit="12"></div>
```

```
<script>
// "Getting" data-attributes using getAttribute
var plant = document.getElementById("strawberry-plant");
var fruitCount = plant.getAttribute("data-fruit"); // fruitCount = "12"

// "Setting" data-attributes using setAttribute
plant.setAttribute("data-fruit","7"); // Pesky birds
</script>
```

Este método funcionará en todos los navegadores modernos, pero no es la manera en la que los *data attributes* deben ser utilizados. La mejor manera para lograr lo mismo es mediante el acceso a la propiedad `dataset` de un elemento. Al utilizar este método, en lugar de utilizar el nombre del atributo completo, se puede prescindir del prefijo `data-` y referirse a los atributos de datos personalizados utilizando directamente los nombres que se han asignado.

```
<div id="sunflower" data-leaves="47" data-plant-height="2.4m"></div>
```

```
<script>
// "Getting" data-attributes using dataset
var plant = document.getElementById("sunflower");
var leaves = plant.dataset.leaves; // leaves = 47;

// "Setting" data-attributes using dataset
var tallness = plant.dataset.plantHeight; // "plant-height" -> "plantHeight"
plant.dataset.plantHeight = "3.6m";  // Cracking fertiliser
</script>
```

Si en algún momento un atributo `data-` específico ya no es necesario, es posible eliminarlo por completo del elemento DOM estableciendo un valor nulo.

```
plant.dataset.leaves = null;
```

En conclusión, los *data attributes* personalizados son una buena manera de simplificar el

almacenamiento de datos de la aplicación en las páginas web.

Ejercicio 5

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 6 MULTIMEDIA

Hasta hace no mucho tiempo, la tecnología *Flash* era el dominador indiscutible en el campo multimedia de la web. Gracias a esta tecnología es relativamente sencillo transmitir audio y vídeo a través de la red, y realizar animaciones que de otra manera sería imposible. Prácticamente todos los navegadores tienen incorporado un *plugin* para la reproducción de archivos *flash*, por lo que la elección era clara. **Entonces, ¿por qué una necesidad de cambio?**

Hasta ahora, para incluir un elemento multimedia en un documento, se hacía uso del elemento `<object>`, cuya función es incluir un elemento externo genérico. Debido a la incompatibilidad entre navegadores, se hacía también necesario el uso del elemento `<embed>` y duplicar una serie de parámetros. El resultado era un código de este estilo:

```
<object width="425" height="344">
    <param name="movie"
        value="http://www.youtube.com/v/9sEI1AUFJKw&hl=en_GB&fs=1&"></param>
    <param name="allowFullScreen" value="true"></param>
    <param name="allowscriptaccess" value="always"></param>
    <embed src="http://www.youtube.com/v/9sEI1AUFJKw&hl=en_GB&fs=1&"
        type="application/x-shockwave-flash"
        allowscriptaccess="always"
        allowfullscreen="true" width="425" height="344"></embed>
</object>
```

Dejando de lado que el código es *poco amigable*, en este caso tenemos el problema que el navegador tiene que transmitir el vídeo a un plugin instalado en el navegador, con la esperanza que el usuario tenga instalada la versión correcta o tenga permisos para hacerlo, y muchos otros requisitos que pueden ser necesarios. Los *plugins* pueden causar que el navegador o el sistema se comporte de manera inestable, o incluso podemos crear una inseguridad a usuarios sin conocimientos técnicos, a los que se pide que descarguen e instalen un nuevo software.

De alguna manera, podemos estar creando una barrera para ciertos usuarios, algo que **no es para nada deseable**.

6.1 VÍDEO

Una de las mayores ventajas del elemento `<video>` (y `<audio>`) de HTML5, es que, finalmente, están totalmente integrados en la web. Ya no es necesario depender de *software* de terceros, y esto es una gran ventaja.

Así que ahora, el elemento `<video>` pueden personalizarse a través de estilos CSS. Se puede cambiar su tamaño y animarlo con transiciones CSS, por ejemplo. Podemos acceder a sus propiedades a través de JavaScript, transformarlos y mostrarlos en un `<canvas>`. Y lo mejor de todo, es que podemos manipularlos con total libertad, ya que pertenecen al estándar. Ya no se ejecutan en una *caja negra* a la que no teníamos acceso.

A pesar de que la etiqueta `<video>` se presenta como una alternativa de *flash*, sus funcionalidades van mucho más allá, como veremos a continuación.

6.1.1 MARCADO

Para hacer funcionar el vídeo en HTML, es suficiente con incluir el siguiente marcado, de manera similar que lo hacemos con las imágenes:

```
<video src="movie.webm"> </video>   <!-- Esto funciona en un mundo ideal -->
```

Sin embargo, este ejemplo realmente no hace nada por el momento. Lo único que se muestra es el primer fotograma de la película. Esto es así porque no le hemos dicho al navegador que inicie el vídeo, ni le hemos mostrado al usuario ningún tipo de control para reproducir o pausar el vídeo.

6.1.2 `AUTOPLAY`

Si bien podemos indicar al navegador que reproduzca el vídeo de manera automática una vez se haya cargado la página, en realidad no es una buena práctica, ya que a muchos usuarios les parecerá una práctica muy intrusiva. Por ejemplo, los usuarios de dispositivos móviles, probablemente no querrán que el vídeo se reproduzca sin su autorización, ya que se consume ancho de banda sin haberlo permitido explícitamente. No obstante, la manera de hacerlo es la siguiente:

```
<video src="movie.webm" autoplay>
    <!-- Your fallback content here -->
</video>
```

6.1.3 `CONTROLS`

Proporcionar controles es aproximadamente un 764% mejor que reproducir el vídeo de manera automática. La manera de indicar que se muestren los controles es la siguiente:

```
<video src="movie.webm" controls>
    <!-- Your fallback content here -->
</video>
```

Naturalmente, y al igual que ocurre con los campos de formulario, los navegadores

muestran los controles de manera diferente, ya que la especificación no indica qué aspecto deben tener. De todas maneras, independientemente del aspecto, todos ellos coinciden en los controles a mostrar: reproducir/pausa, una barra de progreso y un control de volumen. Normalmente, los navegadores esconden los controles, y solamente aparecen al mover el ratón sobre el vídeo al utilizar el teclado para controlar el vídeo.



Figura 6.1 Controles de vídeo en Google Chrome

6.1.4 POSTER

El atributo `poster` indica la imagen que el navegador debe mostrar mientras el vídeo se está descargando, o hasta que el usuario reproduce el vídeo. Esto elimina la necesidad de mostrar una imagen externa que después hay que eliminar con JavaScript. Si no se indica este atributo, el navegador muestra el primer fotograma del vídeo, que puede no ser representativo del vídeo que se va a reproducir.



Figura 6.2 Fotograma `poster` del vídeo anterior

6.1.5 `MUTED`

El atributo `muted`, permite que el elemento multimedia se reproduzca inicialmente sin sonido, lo que requiere una acción por parte del usuario para recuperar el volumen. En este ejemplo, el vídeo se reproduce automáticamente, pero sin sonido:

```
<video src="movie.webm" controls autoplay loop muted>
    <!-- Your fallback content here -->
</video>
```

6.1.6 `HEIGHT`, `WIDTH`

Los atributos `height` y `width` indican al navegador el tamaño del vídeo en pixels. Si no se indican estas medidas, el navegador utiliza las medidas definidas en el vídeo de origen, si están disponibles. De lo contrario, utiliza las medidas definidas en el fotograma poster, si están disponibles. Si ninguna de estas medidas está disponible, el ancho por defecto es de `300 pixels`.

Si únicamente se especifica una de las dos medidas, el navegador automáticamente ajusta la medida de la dimensión no proporcionada, conservando la proporción del vídeo.

Si por el contrario, se especifican las dos medidas, pero no coinciden con la proporción del vídeo original, el vídeo no se deforma a estas nuevas dimensiones, sino que se muestra en formato letterbox manteniendo la proporción original.

6.1.7 `LOOP`

El atributo `loop` indica que el vídeo se reproduce de nuevo una vez que ha finalizado su reproducción.

6.1.8 `PRELOAD`

Es posible indicar al navegador que comience la descarga del vídeo antes de que el usuario inicie su reproducción.

```
<video src="movie.webm" controls preload>
    <!-- Your fallback content here -->
</video>
```

Existen tres valores definidos para `preload`. Si no indicamos uno en concreto, es el propio navegador el que decide qué hacer. Por ejemplo, en un dispositivo móvil, el comportamiento por defecto es no realizar ninguna descarga hasta que el usuario lo haya indicado. Es importante recordar que un desarrollador no puede controlar el comportamiento de un navegador: `preload` es un *consejo*, no un comando. El navegador tomará una decisión en función del dispositivo, las condiciones de la red y otros factores.

- `preload=auto`: se sugiere al navegador que comience la descarga.
- `preload=none`: se sugiere al navegador que no comience la descarga hasta que lo indique el usuario.
- `preload=metadata`: este estado sugiere al navegador que cargue los metadatos (dimensiones, fotogramas, duración...), pero no descarga nada más hasta que el usuario lo indique.

6.1.9 `SRC`

Al igual que en elemento `<img>`, el atributo `src` indica la localización del recurso, que el navegador debe reproducir si el navegador soporta el *codec* o formato específico. Utilizar un único atributo `src` es únicamente útil y viable en entornos totalmente controlados, donde conocemos el navegador que accede al sitio web y los *codecs* que soporta.

Sin embargo, como no todos los navegadores pueden reproducir los mismos formatos, en entornos de producción debemos especificar más de una fuente de vídeo.

6.2 CODECS, LA NUEVA GUERRA

En los primeros borradores de la especificación de HTML5, se indicaba que se debía de ofrecer soporte para al menos dos *codecs* multimedia: Ogg Vorbis para audio y Ogg Theora para vídeo. Sin embargo, estos requisitos fueron eliminados después de que Apple y Nokia se opusieran, de modo que la especificación no recomendase ningún *codec* en concreto. Esto ha creado una situación de fragmentación, con diferentes

navegadores optando por diferentes formatos, basándose en sus ideologías o convicciones comerciales.

Actualmente, hay dos *codecs* principales que debemos tener en cuenta: el nuevo formato WebM, construido sobre el formato VP8 que Google compró y ofrece de manera libre, y el formato MP4, que contiene el *codec* propietario H.264.

	WEBM	MP4	OGV
Opera	Sí	No	Sí
Firefox	Sí	Sí	Sí
Chrome	Sí	No	Sí
IE9+	No	Sí	No
Safari	No	Sí	No

WebM funciona en IE9+ y Safari si el usuario ha instalado los codec de manera manual.

Por lo tanto, la mejor solución en estos momentos es ofrecer tanto el formato libre WebM, como el propietario H.264.

6.2.1 MULTIPLES ELEMENTOS `<SOURCE>`

Para poder ofrecer ambos formatos, primeramente debemos codificarlos por separado. Existen diversas herramientas y servicios on-line para realizar esta tarea, pero quizás el más conocido sea Miro Video Converter. Este software, disponible para Windows y Mac, nos permite convertir los vídeos en formato Theora, o H.264 (y muchos otros) optimizados para diferentes tipos de dispositivos como iPhone, Android, PS2, etc.

Una vez dispongamos el vídeo en los distintos formatos, es necesario indicar todas las localizaciones de estos formatos, para que sea el navegador el que decida que formato reproducir. Evidentemente, no podemos especificarlos todos dentro del atributo `scr`, por lo que tendremos que hacerlo de manera separada utilizando el elemento `<source>`.

```
<video controls>
    <source src="leverage-a-synergy.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
    <source src="leverage-a-synergy.webm" type='video/webm; codecs="vp8, vorbis"'>
    <p>Your browser doesn't support video.
        Please download the video in <a href="leverage-a-synergy.webm">webM</a>
        or <a href="leverage-a-synergy.mp4">MP4</a> format.
    </p>
</video>
```

6.2.2 MEDIA QUERIES PARA `VIDEO`

Los ficheros de vídeo tienden a ser pesados, y enviar un vídeo en alta calidad a un dispositivo con un tamaño de pantalla reducido es algo totalmente ineficiente. No hay ningún inconveniente en hacerlo, pero si comprimimos el vídeo y disminuimos sus dimensiones, conseguiremos reducir su tamaño (en `MB`), algo de agradecer en entornos móviles y que dependen de una conexión de datos.

HTML5 permite utilizar el atributo `media` en el elemento `<source>`, ofreciendo la misma funcionalidad que los `Media Queries` en CSS3. Por lo tanto, podemos consultar al navegador por el ancho de la pantalla, la relación de aspecto, colores, etc, y definir el video correcto según las características del dispositivo.

```
<video controls>
    <source src="hi-res.mp4" media="(min-device-width: 800px)">
    <source src="lo-res.mp4">
</video>
```

6.3 API MULTIMEDIA

Los elementos multimedia `<video>` y `<audio>` ofrecen un API JavaScript muy completo y fácil de utilizar. Los eventos y métodos de los elementos de audio y vídeo son exactamente los mismos, su única diferencia se da en los atributos. A continuación se muestra una tabla con el API actual:

Atributos	Métodos	Eventos
error state	load()	loadstart
error	canPlayType(type)	progress
network state	play()	suspend
src	pause()	abort
currentSrc	addTrack(label, kind, language)	error
networkState		emptied
preload		stalled
buffered		play
ready state		pause
readyState		loadedmetadata
seeking		loadeddata
controls		waiting

controls	playing
volume	canplay
muted	canplaythrough
tracks	seeking
tracks	seeked
playback state	timeupdate
currentTime	ended
startTime	ratechange
muted	
paused	
defaultPlaybackRate	
playbackRate	
played	
seekable	
ended	
autoplay	
loop	
width [video only]	
height [video only]	
videoWidth [video only]	
videoHeight [video only]	
poster [video only]	

Gracias a JavaScript y a este nuevo API, tenemos el control completo sobre los elementos multimedia. Esto significa que podemos crearnos nuestros propios controles, extendiendo los que nos ofrece el navegador. Un simple ejemplo de acceso al API del elemento `video`:

```
video.addEventListener('canplay', function(e) {
    this.volume = 0.4;
    this.currentTime = 10;
    this.play();
```

```
}, false);
```

6.4 FULLSCREEN VIDEO

Flash ha ofrecido un modo de pantalla completa durante muchos años a la que muchos navegadores se han *resistido*. La razón principal es la seguridad; ya que si se fuerza a una aplicación o web a funcionar a pantalla completa, el usuario pierde el control del propio navegador, la barra de tareas y controles estándar del sistema operativo. Puede que el usuario no sepa después volver del modo de pantalla completa, o puede haber problemas de seguridad relacionados, como la simulación del sistema operativo, petición de password, etc.

Este nuevo API establece un único elemento *full-screen*. Está pensado para imágenes, video y juegos que utilizan el elemento `canvas`. Una vez que un elemento pasa a pantalla completa, aparece un mensaje de forma temporal para informar al usuario de que puede presionar la tecla `ESC` en cualquier momento para volver a la ventana anterior.

Las principales propiedad, métodos y estilos son:

- `element.requestFullScreen()`: hace que un elemento individual pase a pantalla completa.

```
document.getElementById("myvideo").requestFullScreen().
```

- `document.cancelFullScreen()`: sale del modo pantalla completa y vuelve a la vista del documento.
- `document.fullScreen`: devuelve `true` si el navegador está en pantalla completa.
- `:full-screen`: se trata de una pseudo-clase CSS que se aplica a un elemento cuando está en modo pantalla completa.

Además, podemos modificar los estilos del elemento utilizando CSS:

```
#myelement
{
    width: 500px;
}
#myelement:full-screen
{
    width: 100%;
}
#myelement:full-screen img
{
    width: 100%;
}
```

6.5 AUDIO

El elemento multimedia `audio` es muy similar en cuanto a funcionalidad al elemento `video`. La principal diferencia existe al indicar el atributo `controls` o no. Si lo especificamos, el elemento se mostrará en la página juntamente con los controles. Si no lo hacemos, el audio se reproducirá, pero no existirá ningún elemento visual en el documento. Por supuesto, el elemento existirá en el DOM y tendremos acceso completo a su API desde JavaScript.

6.5.1 MARCADO

Para hacer funcionar el audio en HTML, al igual que con el video es suficiente con incluir lo siguiente:

```
<audio src="audio.mp3">
</audio>
```

Los formatos soportados por los navegadores son los siguientes:

	MP3	MP4	WAV	OGG
Opera	No	No	Sí	Sí
Firefox	No	No	Sí	Sí
Chrome	Sí	Sí	Sí	Sí
IE9+	Sí	Sí	No	No
Safari	Sí	Sí	Sí	No

Por lo tanto, la mejor solución en estos momentos es ofrecer tanto el formato libre OGG, como el propietario MP3, marcado de la siguiente manera:

```
<audio controls>
    <source src="audio.ogg" type="audio/ogg">
    <source src="audio.mp3" type="audio/mpeg">
</audio>
```

Ejercicio 6

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 7 CANVAS

El elemento `canvas` proporciona un API para dibujar líneas, formas, imágenes, texto, etc en 2D, sobre el *lienzo* que del elemento. Este API ya está siento utilizado de manera exhaustiva, en la creación de fondos interactivos, elementos de navegación, herramientas de dibujado, juegos o emuladores. Éste elemento `canvas` es uno de los elementos que cuenta con una de las mayores especificaciones dentro de HTML5. De hecho, el API de dibujado en 2D se ha separado en un documento a parte.

Un ejemplo de lo que se puede llegar a crear, es la recreación del programa MS Paint incluido en Windows 95.

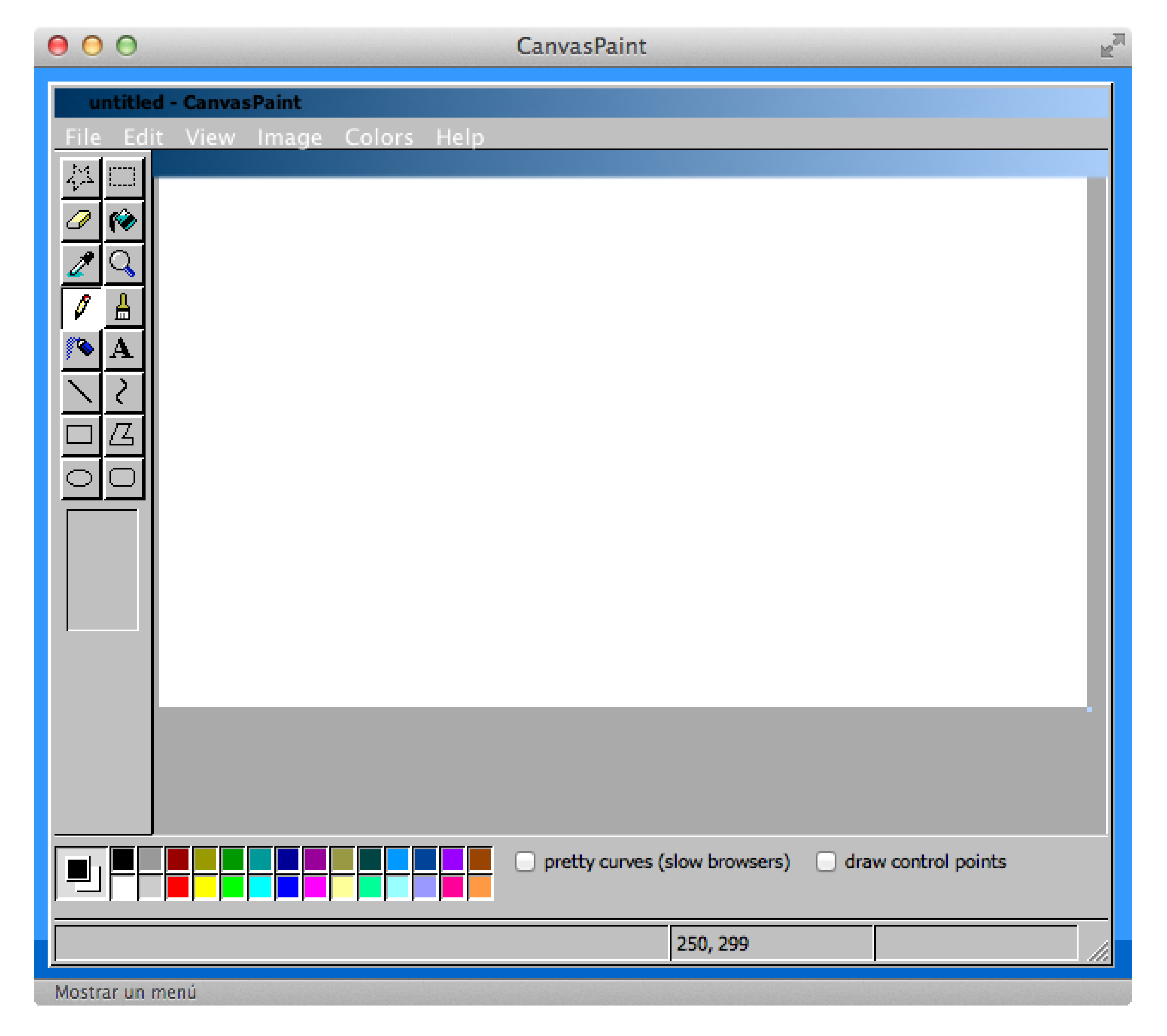


Figura 7.1 MS Paint desarrollado sobre canvas

7.1 ELEMENTOS BÁSICOS

Para comenzar a utilizar el elemento `canvas`, debemos colocarlo en el documento. El marcado es extremadamente sencillo:

```
<canvas id="tutorial" width="150" height="150"></canvas>
```

Éste nos recuerda mucho al elemento `img`, pero sin los atributos `src` y `alt`. En realidad, el elemento `canvas` solamente tiene los dos atributos mostrados en el ejemplo anterior: `width` y `height`, ambos opcionales y que pueden establecerse mediante las propiedades DOM. Cuando estos dos atributos no se especifican, el lienzo (*canvas*) inicial será de `300px` de ancho por `150px` de alto. Este elemento, además y como muchos otros, pueden modificarse utilizando CSS. Podemos aplicar cualquier estilo, pero las reglas afectarán al elemento, no a lo dibujado en el lienzo.

En la actualidad, todos los navegadores son compatibles con el elemento `canvas`, la diferencia entre ellos radica en qué funcionalidades del API han implementado.

Ahora que el elemento `canvas` está definido en el documento, la manera de dibujar en él es a través de JavaScript. El primer paso es obtener el **contexto** de dibujado. `<canvas>` crea una superficie de dibujo de tamaño fijo que expone uno o más **contextos** de representación, que se utilizan para crear y manipular el contenido mostrado. En el contexto de representación 2D, (existe otro contexto de representación en 3D, llamado WebGL), el `canvas` está inicialmente en blanco y, para mostrar algo, es necesario el acceso de un `script` al contexto de representación para que pueda dibujar en él. El método DOM `getContext` sirve para obtener el contexto de representación y sus funciones de dibujo.

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
```

El siguiente ejemplo dibujaría dos rectángulos que se cruzan, uno de los cuales tiene transparencia *alfa*.

```
<html>
  <head>
    <script type="application/javascript">
      window.onload = function() {
        var canvas = document.getElementById("canvas");
        if (canvas.getContext) {
          var ctx = canvas.getContext("2d");
          ctx.fillStyle = "rgb(200,0,0)";
          ctx.fillRect (10, 10, 55, 50);
          ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
          ctx.fillRect (30, 30, 55, 50);
        }
```

```
    };
  </script>
 </head>
 <body>
   <canvas id="canvas" width="150" height="150"></canvas>
 </body>
</html>
```

Figura 7.2 Rectángulos semitransparentes en canvas

7.2 DIBUJAR FORMAS

Para empezar a dibujar formas, es necesario hablar primero de la cuadrícula del `canvas` o espacio de coordenadas. Normalmente, una unidad en la cuadrícula corresponde a un px en el lienzo. El punto de origen de esta cuadrícula se coloca en la esquina superior izquierda (coordenadas(0,0)). Todos los elementos se colocan con relación a este origen.

Desgraciadamente, `canvas` solamente admite una forma primitiva: los rectángulos, por lo que el resto de las formas deberán crearse mediante la combinación de una o más funciones.

Existen tres funciones que dibujan un rectángulo en el lienzo:

- `fillRect(x,y,width,height)`: dibuja un rectángulo relleno .
- `strokeRect(x,y,width,height)`: dibuja un contorno rectangular .
- `clearRect(x,y,width,height)`: borra el área especificada y hace que sea totalmente transparente.

Cada una de estas funciones tiene los mismos parámetros. `x` e `y` especifican la posición en el lienzo. `width` es la anchura y `height` la altura.

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
ctx.fillRect(25,25,100,100);
```

```
ctx.clearRect(45,45,60,60);
ctx.strokeRect(50,50,50,50);
```

La función `fillRect` dibuja un gran cuadrado negro de 100x100 px. La función `clearRect` elimina un cuadrado de 60x60 px del centro y finalmente el `strokeRect` dibuja un contorno rectangular de 50x50 px en el interior del cuadrado despejado.

A diferencia de las funciones de rutas que veremos en la siguiente sección, las tres funciones de rectángulo se dibujan inmediatamente en el lienzo.

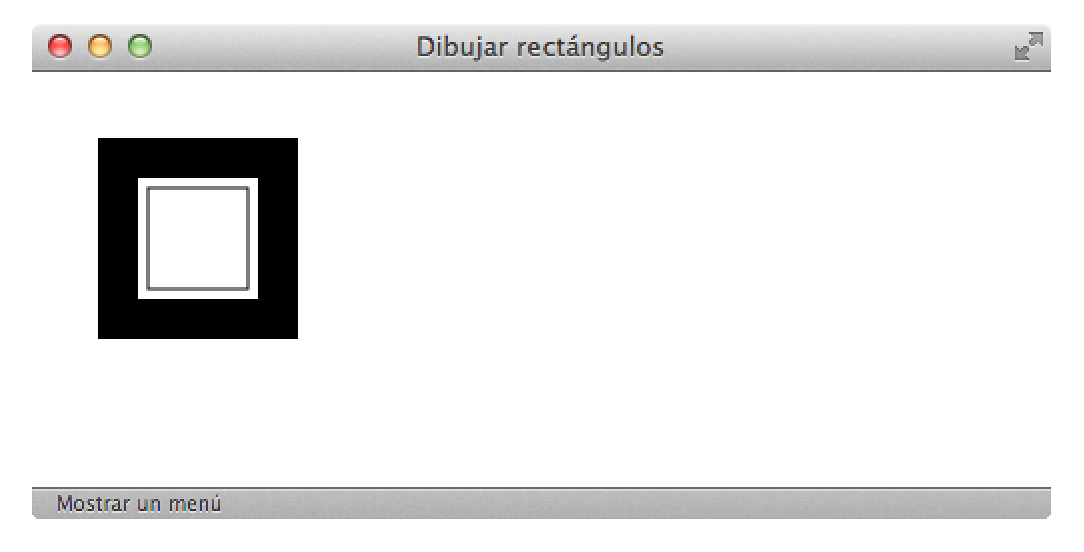


Figura 7.3 Dibujado de rectángulos en canvas

7.3 RUTAS

Gracias al API 2D, es posible movernos a través del canvas y dibujar líneas y formas. Las rutas son utilizadas para dibujar formas (líneas, curvas, polígonos, etc) que de otra forma no podríamos conseguir.

El primer paso para crear una ruta es llamar al método `beginPath`. Internamente, las rutas se almacenan como una lista de subrutas (líneas, arcos, etc.) que, en conjunto, forman una figura. Cada vez que se llama a este método, la lista se pone a cero y podemos empezar a dibujar nuevas formas. El paso final sería llamar al método `closePath`: este método intenta cerrar la forma trazando una línea recta desde el punto actual hasta el inicial. Si la forma ya se ha cerrado o hay solo un punto en la lista, esta función no hace nada.

```
var canvas  = document.getElementById('tutorial');
var context = canvas.getContext('2d');
```

```
context.beginPath();
//... path drawing operations
context.closePath();
```

El siguiente paso es dibujar la forma como tal. Para ello, disponemos de algunas

funciones de dibujado de líneas y arcos, que especifican las rutas a dibujar.

7.3.1 MÉTODO `LINETO`

Para dibujar líneas rectas utilizamos el método `lineTo`. Este método toma dos argumentos `x` e `y`, que son las coordenadas del punto final de la línea. El punto de partida depende de las rutas anteriores.

En el siguiente ejemplo se dibujan dos triángulos, uno relleno y el otro únicamente trazado. En primer lugar se llama al método `beginPath` para iniciar una nueva ruta. A continuación, utilizamos el método `moveTo` para mover el punto de partida hasta la posición deseada. Finalmente se dibujan dos líneas que forman dos lados del triángulo. Al llamar al método `closePath`, éste traza una línea al origen, completando el triángulo.

```
// Triángulo relleno
ctx.beginPath();
ctx.moveTo(25,25);
ctx.lineTo(105,25);
ctx.lineTo(25,105);
ctx.closePath();
ctx.fill();
```

```
// Triángulo trazado
ctx.beginPath();
ctx.moveTo(125,125);
ctx.lineTo(125,45);
ctx.lineTo(45,125);
ctx.closePath();
ctx.stroke();
```

En ambos casos utilizamos dos funciones de *pintado* diferentes: `stroke` y `fill`. `Stroke` se utiliza para dibujar una forma con contorno, mientras que `fill` se utiliza para pintar una forma sólida.

7.3.2 ARCOS

Para dibujar arcos o círculos se utiliza el método `arc` (la especificación también describe el método `arcTo`). Este método toma cinco parámetros: `x` e `y`, el radio, `startAngle` y `endAngle` (que definen los puntos de inicio y final del arco en radianes) y `anticlockwise` (un valor booleano que, cuando tiene valor `true` dibuja el arco de modo levógiro y viceversa cuando es `false`).

Un ejemplo algo más complejo que los anteriores utilizando el método `arc` sería:

```
for(var i=0;i<4;i++){
    for(var j=0;j<3;j++){
```

```
        ctx.beginPath();
        var x              = 25+j*50;       // coordenada x
        var y              = 25+i*50;       // coordenada y
        var radius         = 20;            // radio del arco
        var startAngle     = 0;             // punto inicial del círculo
        var endAngle       = Math.PI+(Math.PI*j)/2; // punto final
        var anticlockwise  = i%2==0 ? false : true;
        ctx.arc(x,y,radius,startAngle,endAngle, anticlockwise);
        if (i>1){
            ctx.fill();
        } else {
            ctx.stroke();
        }
    }
}
```

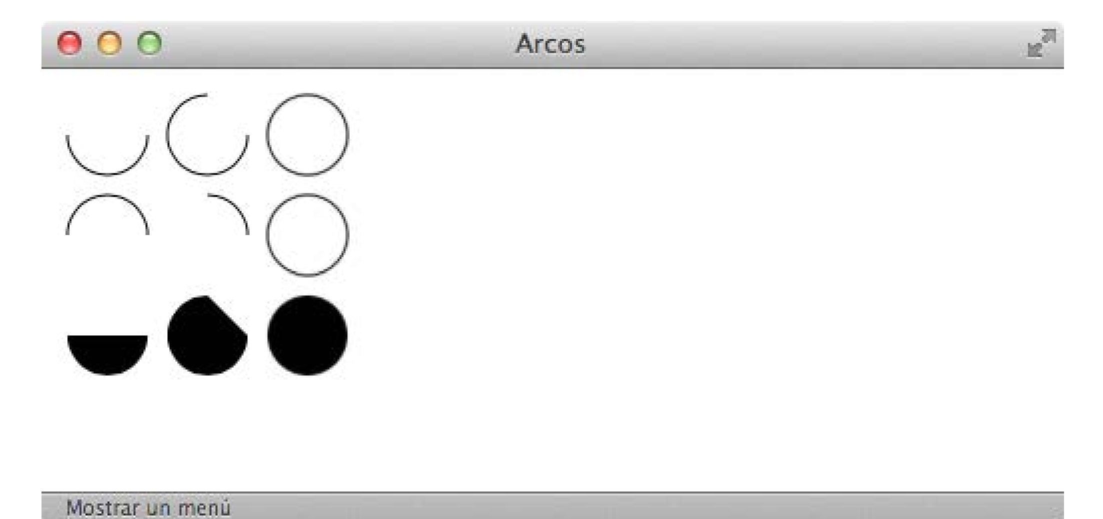


Figura 7.4 Dibujado de arcos en canvas

> ***Nota***
>
> *Los métodos `arc`, `bezier` y `quadratic` utilizan radianes, pero si preferimos trabajar en grados, es necesario convertirlo a radianes. Esta el la operación que tenemos que llevar a cabo:*
>
> `var radians = degrees * Math.PI / 180;`

7.3.3 MÉTODO `MOVETO`

Disponemos de la función `moveTo`, que en realidad, aunque no dibuja nada, podemos imaginárnosla como 'si levantas un lápiz desde un punto a otro en un papel y lo colocas en el siguiente'. Esta función es utilizada para colocar el punto de partida en otro lugar o para dibujar rutas inconexas.

```
ctx.beginPath();
ctx.arc(75,75,50,0,Math.PI*2,true);  // círculo exterior
```

```
ctx.closePath();
ctx.fill();
```

```
ctx.fillStyle = '#FFF';
ctx.beginPath();
ctx.arc(75,75,35,0,Math.PI,false);   // boca (dextrógiro)
ctx.closePath();
ctx.fill();
```

```
ctx.moveTo(65,65);
ctx.beginPath();
ctx.arc(60,65,5,0,Math.PI*2,true);  // ojo izquierdo
ctx.closePath();
ctx.fill();
```

```
ctx.moveTo(95,65);
ctx.beginPath();
ctx.arc(90,65,5,0,Math.PI*2,true);  // ojo derecho
ctx.closePath();
ctx.fill();
```

Cara sonriente :)

Mostrar un menú

Figura 7.5 Dibujar una cara sonriente en canvas

7.4 COLORES

Si queremos aplicar colores a una forma, hay dos características importantes que podemos utilizar: `fillStyle` y `strokeStyle`.

```
fillStyle = color
strokeStyle = color
```

`strokeStyle` se utiliza para configurar el color del contorno de la forma y `fillStyle` es para el color de relleno. Color puede ser una cadena que representa un valor de color CSS, un objeto degradado o un objeto modelo.

```
// todos ellos configuran fillStyle a 'naranja' (orange)
ctx.fillStyle = "orange";
ctx.fillStyle = "#FFA500";
ctx.fillStyle = "rgb(255,165,0)";
ctx.fillStyle = "rgba(255,165,0,1)";
```

Ejemplo de `fillStyle`:

```
function draw() {
    for (var i=0;i<6;i++){
        for (var j=0;j<6;j++){
            ctx.fillStyle = 'rgb(' + Math.floor(255-42.5*i) + ','
                                    + Math.floor(255-42.5*j) + ',0)';
            ctx.fillRect(j*25,i*25,25,25);
        }
    }
}
```

Ejemplo de `strokeStyle`:

```
function draw() {
    for (var i=0;i<6;i++){
        for (var j=0;j<6;j++){
            ctx.strokeStyle = 'rgb(0,' + Math.floor(255-42.5*i)
                                        + ',' +  ath.floor(255-42.5*j) + ')';
            ctx.beginPath();
            ctx.arc(12.5+j*25,12.5+i*25,10,0,Math.PI*2,true);
            ctx.stroke();
        }
    }
}
```

7.5 DEGRADADOS Y PATRONES

A través del contexto, es posible generar degradados lineales, radiales o rellenos a través de patrones, que pueden ser utilizados en el método `fillStyle` del canvas. Los degradados funcionan de una manera similar a los definidos en CSS3, donde se especifican el inicio y los pasos de color para el degradado.

Los patrones, por otra parte, permiten definir una imagen como origen y especificar el patrón de repetido, de nuevo de manera similar a como se realizaba con la propiedad `background-image` de CSS. Lo que hace interesante al método `createPattern` es que como origen podemos utilizar una imagen, un canvas o un elemento de vídeo.

Un simple gradiente se crea de la siguiente manera:

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
var gradient = ctx.createLinearGradient(0, 0, 0, canvas.height);
```

```
gradient.addColorStop(0, '#fff');
gradient.addColorStop(1, '#000');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

El código anterior creamos un degradado lineal al que aplicamos dos pasos de color. Los argumentos de `createLinearGradient` son el punto de inicio del degradado (`x1` e `y1`) y el punto final del degradado (`x2` e `y2`). En este caso el degradado comienza en la esquina superior izquierda, y termina en la esquina inferior izquierda. Utilizamos este degradado para pintar el fondo de un rectángulo.



Figura 7.6 Degradado lineal

Los degradados radiales son muy similares, con la excepción que definimos el radio después de cada coordenada:

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
gradient = ctx.createRadialGradient(canvas.width/2,
                                    canvas.height/2,
                                    0,
                                    canvas.width/2,
                                    canvas.height/2,
                                    150);
gradient.addColorStop(0, '#fff');
gradient.addColorStop(1, '#000');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, canvas.width, canvas.width);
```

La única diferencia es la manera en la que se ha creado el degradado. En este ejemplo, el primer punto del degradado se define en el centro del canvas, con radio cero. El siguiente punto se define con un radio de `150px` pero su origen es el mismo, lo que produce un degradado circular.

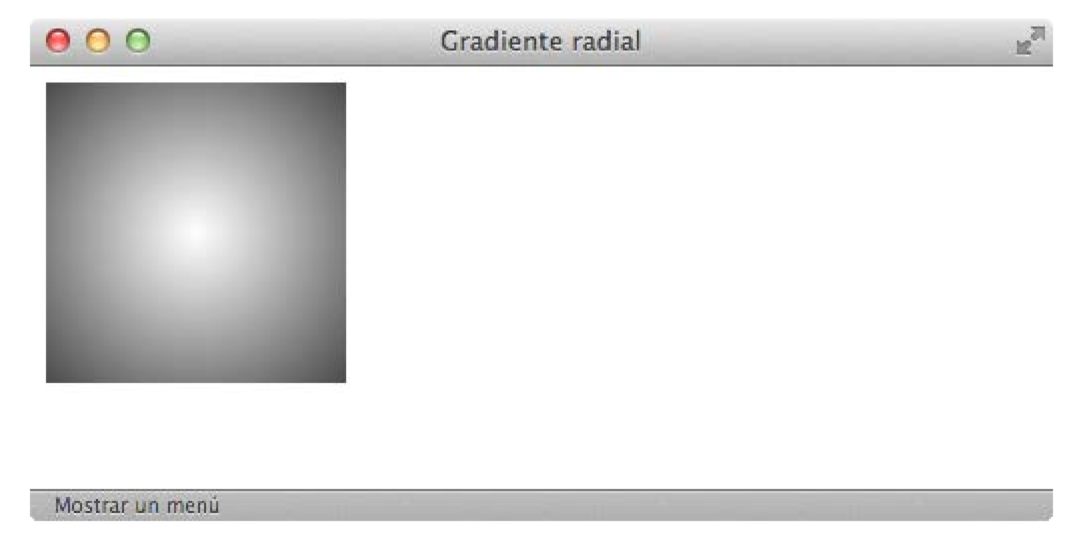


Figura 7.7 Degradado radial

Los patrones son incluso más sencillos de utilizar. Es necesaria una fuente (como una imagen, un canvas o un elemento de vídeo) y posteriormente utilizamos esta fuente en el método `createPattern` y el resultado de éste en el método `fillStyle`. La única consideración a tener en cuenta es que, al utilizar elementos de imagen o vídeo, éstos tienen que haber terminado de cargarse para poder utilizarlos. En el siguiente ejemplo, expandimos el canvas para que ocupe toda la ventana, y cuando se carga la imagen, la utilizamos como patrón de repetición.

```
var canvas = document.getElementById('tutorial');
var img = document.createElement('img');
var ctx = canvas.getContext('2d');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
```

```
img.onload = function () {
    ctx.fillStyle = ctx.createPattern(this, 'repeat');
    ctx.fillRect(0, 0, canvas.width, canvas.height);
};
img.src = 'avatar.jpg';
```

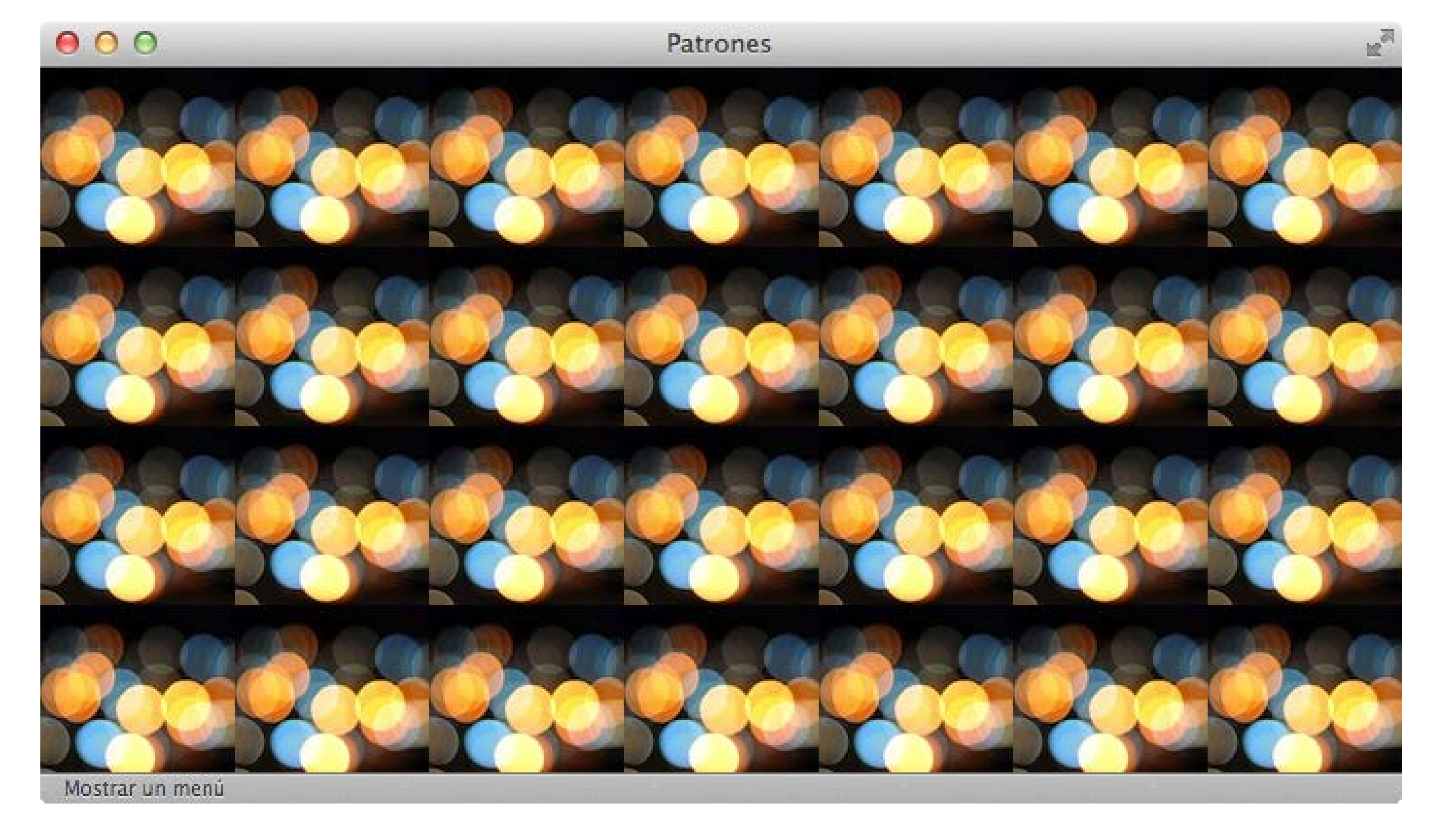


Figura 7.8 Uso de patrones en canvas

7.6 TRANSPARENCIAS

Además de dibujar formas opacas en el lienzo, también podemos dibujar formas semitransparentes. Esto se hace mediante el establecimiento de la propiedad `globalAlpha` o podríamos asignar un color semitransparente al trazo y/o al estilo de relleno.

```
globalAlpha = transparency value
```

Esta propiedad aplica un valor de transparencia a todas las formas dibujadas en el lienzo y puede ser útil si deseas dibujar un montón de formas en el lienzo con una transparencia similar; ya que debido a que las propiedades `strokeStyle` y `fillStyle` aceptan valores de color CSS3, podemos utilizar la siguiente notación para asignarles un color transparente.

```
function draw() {
    // dibujar fondo
    ctx.fillStyle = '#FD0';
    ctx.fillRect(0,0,75,75);
    ctx.fillStyle = '#6C0';
    ctx.fillRect(75,0,75,75);
    ctx.fillStyle = '#09F';
    ctx.fillRect(0,75,75,75);
    ctx.fillStyle = '#F30';
    ctx.fillRect(75,75,150,150);
```

```
    ctx.fillStyle = '#FFF';
    // establecer valor de transparencia
    ctx.globalAlpha = 0.2;
    // Dibujar círculos semitransparentes
    for (var i=0;i<7;i++){
        ctx.beginPath();
        ctx.arc(75,75,10+10*i,0,Math.PI*2,true);
        ctx.fill();
    }
}
```

7.7 TRANSFORMACIONES

Al igual que tenemos la posibilidad mover el *lápiz* por el canvas con el método `moveTo`, podemos definir algunas transformaciones como rotación, escalado, transformación y traslación (similares a las conocidas de CSS3).

7.7.1 MÉTODO `TRANSLATE`

Éste método traslada el centro de coordenadas desde su posición por defecto (0, 0) a la posición indicada.

```
ctx.translate(x, y);
```

7.7.2 MÉTODO `ROTATE`

Éste método inicia la rotación desde su posición por defecto (0,0). Si se rota el canvas desde esta posición, el contenido podría desaparecer por los límites del lienzo, por lo que es necesario definir un nuevo origen para la rotación, dependiendo del resultado deseado.

```
ctx.rotate(angle);
```

Éste método sólo precisa tomar un parámetro, que es el ángulo de rotación que se aplicará al marco. Este parámetro es una rotación dextrógira medida en radianes.

```
function draw() {
    ctx.translate(75,75);
    for (i=1;i<6;i++){
        // Desplazarse or los anillos (desde dentro hacia fuera)
        ctx.save();
        ctx.fillStyle = 'rgb('+(51*i)+','+(255-51*i)+',255)';
        for (j=0;j<i*6;j++){
            // dibujar puntos individuales
            ctx.rotate(Math.PI*2/(i*6));
            ctx.beginPath();
            ctx.arc(0,i*12.5,5,0,Math.PI*2,true);
```

```
            ctx.fill();
        }
        ctx.restore();
    }
}
```

7.7.3 MÉTODO `SCALE`

El siguiente método de transformación es el escalado. Se utiliza para aumentar o disminuir las unidades del tamaño de nuestro marco. Este método puede usarse para dibujar formas ampliadas o reducidas.

```
ctx.scale(x, y);
```

`x` e `y` definen el factor de escala en la dirección horizontal y vertical respectivamente. Los valores menores que 1.0 reducen el tamaño de la unidad y los valores mayores que 1.0 aumentan el tamaño de la unidad. Por defecto, una unidad en el área de trabajo equivale exactamente a un píxel. Si aplicamos, por ejemplo, un factor de escalado de 0.5, la unidad resultante será 0.5 píxeles, de manera que las formas se dibujarán a mitad de su tamaño.

7.8 ANIMACIONES

Como estamos utilizando *scripts* para controlar los elementos `canvas`, resulta muy fácil hacer animaciones (interactivas). Sin embargo, el elemento `canvas` no fue diseñado para ser utilizado de esta manera (a diferencia de *flash*) por lo que existen limitaciones.

Probablemente, la mayor limitación es que una vez que se dibuja una forma, se queda de esa manera. Si necesitamos moverla, tenemos que volver a dibujar dicha forma y todo lo que se dibujó anteriormente.

Los pasos básicos a seguir son los siguientes:

1. Borrar el lienzo: a menos que las formas que se dibujen llenen el lienzo completo (por ejemplo, una imagen de fondo), se necesitará borrar cualquier forma que se haya dibujado con anterioridad. La forma más sencilla de hacerlo es utilizando el método `clearRect`.
2. Guardar el estado de `canvas`: si se va a modificar alguna configuración (estilos, transformaciones) que afectan al estado de `canvas`.
3. Dibujar formas animadas: representación del marco.
4. Restaurar el estado de canvas: restaurar el estado antes de dibujar un nuevo marco.

7.8.1 CONTROL

Las formas se dibujan en el lienzo mediante el uso de los métodos `canvas` directamente o llamando a funciones personalizadas. Necesitamos una manera de ejecutar nuestras funciones de dibujo en un período de tiempo. Hay dos formas de controlar una animación como ésta. En primer lugar está las funciones `setInterval` y `setTimeout`, que se pueden utilizar para llamar a una función específica durante un período determinado de tiempo.

```
setInterval (animateShape, 500);
setTimeout (animateShape, 500);
```

El segundo método que podemos utilizar para controlar una animación es el *input* del usuario. Si quisiéramos hacer un juego, podríamos utilizar los eventos de teclado o de ratón para controlar la animación. Al establecer `EventListeners`, capturamos cualquier interacción del usuario y ejecutamos nuestras funciones de animación.

Ejercicio 7

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 8 ALMACENAMIENTO LOCAL

El almacenamiento de datos es fundamental en cualquier aplicación web o de escritorio. Hasta ahora, el almacenamiento de datos en la web se realizaba en el servidor, y era necesario algún tipo de conexión con el cliente para trabajar con estos datos. Con HTML5 disponemos de tres tecnologías que permiten que las aplicaciones almacenen datos en los dispositivos cliente. Según las necesidades de la aplicación, la información puede sincronizarse también con el servidor o permanecer siempre en el cliente. Estas son las posibilidades que tenemos:

- Web Storage: http://www.w3.org/TR/webstorage/. Es el sistema de almacenamiento más simple, ya que los datos se almacenan en parejas de clave/valor. Ampliamente soportado por todos los navegadores.
- Web SQL Database: http://www.w3.org/TR/webdatabase/. Sistema de almacenamiento basado en SQL. La especificación indica que no va a ser mantenido en el futuro, pero actualmente su uso está muy extendido y es soportado por Chrome, Safari y Opera.
- IndexedDB: http://www.w3.org/TR/Indexeddb/. Sistema de almacenamiento basado en objetos. Actualmente soportado por Chrome, Firefox e Internet Explorer.

8.1 WEB STORAGE

Este API de almacenamiento ofrece dos posibilidades para guardar datos en el navegador: `sessionStorage` y `localStorage`. El primero mantiene los datos durante la sesión actual (mientras la ventana o pestaña se mantenga abierta), mientras que el segundo almacena los datos hasta que sean eliminados explícitamente por la aplicación o el usuario. Ambos modos de almacenamiento se encuentran relacionados con el dominio que los ha creado.

- `sessionStorage`: objeto global que mantiene un área de almacenamiento disponible a lo largo de la duración de la sesión de la ventana o pestaña. La sesión persiste mientras que la ventana permanezca abierta y sobrevive a recargas de página. Si se abre una nueva página en una pestaña o ventana, una nueva sesión es inicializada, por lo que no es posible acceder a los datos de otra sesión.
- `localStorage`: el almacenamiento local por su parte, funciona de la misma forma que el almacenamiento de sesión, con la excepción de que son capaces de almacenar los datos por dominio y persistir más allá de la sesión actual, aunque el navegador se cierre o el dispositivo se reinicie.

> ***Nota***
>
> *Cuando hacemos referencia la ventana o pestaña, nos estamos refiriendo al objeto `window`. Una nueva ventana abierta utilizando el método `window.open()`, pertenece a la misma sesión.*

Tanto `sessionStorage` como `localStorage` forman parte del Web Storage, por lo que comparten el mismo API:

```
readonly attribute unsigned long length;
getter DOMString key(in unsigned long index);
getter DOMString getItem(in DOMString key);
setter creator void setItem(in DOMString key, in any data);
deleter void removeItem(in DOMString key);
void clear();
```

Este API hace que sea muy sencillo acceder a los datos. El método `setItem` almacena el valor, y el método `getItem` lo obtiene, como se muestra a continuación:

```
sessionStorage.setItem('twitter', '@starkyhach');
alert( sessionStorage.getItem('twitter') ); // muestra @starkyhach
```

Es importante darse cuenta, que tal y como se indica en el API, el método `getItem` *siempre devuelve un String*, por lo que si intentamos almacenar un objeto, el valor devuelto será "[Object object]". El mismo problema ocurre con los número, por lo que es importante tenerlo en cuenta para evitar posibles errores. Por ejemplo:

```
sessionStorage.setItem('total', 120);
function calcularCosteEnvio(envio) {
    return sessionStorage.getItem('total') + envio;
}

alert(calcularCosteEnvio(25));
```

En este caso, esperamos que el coste total (120) se almacene como número, y al añadir el coste del envío, el resultado sea 145. Pero como `sessionStorage` devuelve un *String*, el resultado no es el esperado sino 12025.

8.1.1 ELIMINANDO DATOS

Disponemos de tres formas de eliminar datos del almacenamiento local: utilizando `delete`, `removeItem` y `clear`. El método `removeItem` toma como parámetro el nombre de la clave a eliminar (el mismo que utilizamos en `getItem` y `setItem`), para eliminar un ítem en particular. Por su parte, el método `clear`, elimina todas las entradas del objeto.

```
sessionStorage.setItem('twitter', '@starkyhach');
```

```
sessionStorage.setItem('flickr', 'starky.hach');
alert( sessionStorage.length );                     // Muestra 2
sessionStorage.removeItem('twitter');
alert( sessionStorage.length );                     // Muestra 1
sessionStorage.clear();
alert( sessionStorage.length );                     // Muestra 0
```

8.1.2 ALMACENANDO ALGO MÁS QUE STRINGS

Una manera de almacenar objetos es utilizando JSON. Como la representación de los objetos en JSON puede realizarse a través de texto, podemos almacenar estas cadenas de texto y recuperarlas posteriormente para convertirlas en objetos.

```
var videoDetails = {
        title : 'Matrix',
        author : ['Andy Wachowski', 'Larry Wachowski'],
        description : 'Wake up Neo, the Matrix has you...',
        rating: '-2'
};

sessionStorage.setItem('videoDetails', JSON.stringify(videoDetails) );
var videoDetails = JSON.parse(sessionStorage.getItem('videoDetails'));
```

8.1.3 EVENTOS DE ALMACENAMIENTO

Una de las funcionalidades más interesantes de Web Storage es que incluye una serie de eventos que nos indican cuándo se ha producido un cambio en los datos almacenados. Estos eventos no se lanzan en la ventana actual donde se han producido los cambios, sino en el resto de ventadas donde los datos pueden verse afectados.

Esto quiere decir que los eventos para `sessionStorage` son lanzados en los `iframe` dentro de la misma página, o en las ventanas abiertas con `window.open()`. Para `localStorage`, todas las ventanas abiertas con el mismo origen (protocolo + host + puerto) reciben los eventos.

Cuando se lanzan los eventos, éstos contienen toda la información asociada con el cambio de datos:

```
StorageEvent {
    readonly DOMString key;
    readonly any oldValue;
    readonly any newValue;
    readonly DOMString url;
    readonly Storage storageArea;
};
```

`storageArea` hace referencia al objeto `sessionStorage` o `localStorage`. Estos eventos se

lanzan dentro del objeto `window`:

```
function handleStorage(event) {
    event = event || window.event; // support IE8
    if (event.newValue === null) { // it was removed
        // Do somthing
    } else {
        // Do somthing else
    }
}
```

```
window.addEventListener('storage', handleStorage, false);
window.attachEvent('storage', handleStorage);
```

Ejercicio 8

Ver enunciado

8.2 WEB SQL

Web SQL es otra manera de almacenar y acceder a datos en el dispositivo. Realmente, no forma parte de la especificación de HTML5, pero es ampliamente utilizado para el desarrollo de aplicaciones web. Como su nombre indica, es una base de datos basada en SQL, más concretamente en SQLite. La utilización del API se resume en tres simples métodos:

- `openDatabase`: abrir (o crear y abrir) una base de datos en el navegador del cliente.
- `transaction`: iniciar una transacción.
- `executeSql`: ejecutar una sentencia SQL.

Como en la mayoría de librearías de JavaScript, el API de Web SQL realiza llamadas diferidas a funciones, una vez completadas las operaciones.

```
transaction.executeSql(sql, [], function () {
    // my executed code lives here
});
```

Debido a la naturaleza de estas llamadas, significa que el API de Web SQL es asíncrono, por lo que es necesario tener cuidado con el orden en el que se ejecutan las sentencias SQL. Sin embargo, las sentencias SQL se encolan y son ejecutadas en orden, por lo que podemos estar tranquilos en ese sentido: podemos crear tablas y tener la seguridad que van a ser creadas antes de acceder a los datos.

8.2.1 CREANDO Y ABRIENDO LA BD

El uso clásico de la API implica abrir (o crear) la base de datos y ejecutar algunas

sentencias SQL. Al abrir la base de datos por primera vez, ésta es creada automáticamente. Es necesario especificar el número de versión de base de datos con el que se desea trabajar, y si no especificamos correctamente éste número de versión, es posible que provoquemos un error del tipo `INVALID_STATE_ERROR`.

```
var db = openDatabase('mydb', '1.0', 'My first database', 2 * 1024 * 1024);
```

La última versión de la especificación incluye un quinto argumento en la función `openDatabase`, pero no es soportado por muchos navegadores. Los valores que pasamos a esta función son los siguientes:

1. El nombre de la base de datos.
2. El número de versión de base de datos con el que deseamos trabajar.
3. Un texto descriptivo de la base de datos.
4. Tamaño estimado de la base de datos, en `bytes`.

El valor de retorno de esta función es un objeto que dispone de un método `transaction`, a través del cual vamos a ejecutar las sentencias SQL.

8.2.2 TRANSACCIONES

Ahora que tenemos la base de datos abierta, podemos crear transacciones para ejecutar nuestras sentencias SQL. La idea de utilizar transacciones, en lugar de ejecutar las sentencias directamente, es la posibilidad de realizar *rollback*. Esto quiere decir, que si la transacción falla por algún motivo, se vuelve al estado inicial, *como si nada hubiese pasado*.

```
db.transaction(function (tx) {
    tx.executeSql('DROP TABLE foo');

    // known to fail - so should rollback the DROP statement
    tx.executeSql('INSERT INTO foo (id, text) VALUES (1, "foobar")');
}, function (err) {
    alert(err.message);
});
```

Una vez listo el objeto transacción (`tx` en el ejemplo), podemos ejecutar sentencias SQL.

8.2.3 EJECUTAR SENTENCIAS SQL

El método `executeSql` es utilizado tanto para sentencias de escritura como de lectura. Incluye protección contra ataques de inyección SQL y proporciona llamadas a métodos (*callback*) para procesar los resultados devueltos por una consulta SQL.

Su sintaxis es la siguiente:

```
db.transaction(function (tx) {
    tx.executeSql(sqlStatement, arguments, callback, errorCallback);
});
```

Donde:

1. `sqlStatement`: indica la sentencia SQL a ejecutar. Como hemos dicho, puede ser cualquier tipo de sentencia; creación de tabla, insertar un registro, realizar una consulta, etc.
2. `arguments`: corresponde con un *array* de argumentos que pasamos a la sentencia SQL. Es recomendable pasar los argumentos a la sentencia de esta manera, ya que el propio método se ocupa de prevenir inyecciones SQL.
3. `callback`: función a ejecutar cuando la transacción se ha realizado de manera correcta. Toma como parámetros la propia transacción y el resultado de la transacción.
4. `erroCallback`: función a ejecutar cuando la transacción se ha producido un error en la sentencia SQL. Toma como parámetros la propia transacción y el error producido.

Un ejemplo de selección de registros sería el siguiente:

```
db.transaction(function (tx) {
    tx.executeSql('SELECT * FROM foo WHERE id = ?', [5],
                  function callback(tx, results) {
                      var len = results.rows.length, i;
                      for (i = 0; i < len; i++) {
                        alert(results.rows.item(i).text);
                      }
                    },
                  function errorCallback(tx, error) {
                    alert(error.message);
                  }
                  );
});
```

La función de *callback* recibe como argumentos la transacción (de nuevo) y un objeto que contiene los resultados. Este objeto contiene una propiedad `rows`, donde `rows.item(i)` contiene la representación de la fila concreta. Si nuestra tabla contiene un campo que se llama *nombre*, podemos acceder a dicho campo de la siguiente manera:

```
results.rows.item(i).nombre
```

8.2.4 CREAR TABLAS

La primera tarea a realizar cuando trabajamos con una base de datos es crear las tablas necesarias para almacenar los datos. Como hemos comentado antes, este proceso se realiza a través de una sentencia SQL, dentro de una transacción. Vamos a crear una base de datos para almacenar *tweets* que posteriormente obtendremos de internet:

```
db = openDatabase('tweetdb', '1.0', 'All my tweets', 2 * 1024 * 1024);
db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE IF NOT EXISTS tweets(id, user, date, text)', []
, getTweets);
});
```

8.2.5 INSERTAR DATOS

Una vez creada la tabla, el siguiente paso es insertar los datos correspondientes. Vamos a suponer que hemos realizado una petición AJAX a la API de Twitter, que nos ha devuelto una serie de tweets que queremos almacenar en nuestra base de datos. La función `getTweets` tendría este aspecto:

```
function getTweets() {
    var tweets = $.ajax({...});
    $.each(tweets, function(tweet) {
        db.transaction(function (tx) {
            var time = (new Date(Date.parse(tweet.created_at))).getTime();
            tx.executeSql('INSERT INTO tweets (id, user, date, text) VALUES (?,
 ?, ?, ?)',
                          [tweet.id, tweet.from_user, time / 1000, tweet.text])
;
        });
    });
}
```

> *Nota*
>
> *Insertando cada tweet en una nueva transacción, nos aseguramos que si se produce un error en alguna de ellas (ya existía el tweet), se van a seguir ejecutando el resto transacciones.*

8.2.6 OBTENER DATOS

Finalmente, una vez que los datos se encuentran almacenados en la base de datos, sólo nos queda consultarlos. Como siempre, ejecutamos las sentencia SQL dentro de una transacción y proveemos la función que procesará los datos obtenidos:

```
db.transaction(function (tx) {
    tx.executeSql('SELECT * FROM tweets WHERE date > ?', [time],
                  function(tx, results) {
                      var html = [], len = results.rows.length;
```

```
                        for (var i = 0; i < len; i++) {
                            html.push('<li>' + results.rows.item(i).text + '</l
i>');
                        }
                        tweetEl.innerHTML = html.join('');
                    });
                });
```

Ejercicio 9

Ver enunciado

8.3 INDEXEDDB

`IndexedDB` no es una base de datos relacional, sino que se podría llamar un *almacén de objetos* ya que en la base de datos que creemos, existen almacenes y en su interior añadimos objetos (como el siguiente):

```
{
    id:21992,
    nombre: "Memoria RAM"
}
```

En `IndexedDB`, al igual que en Web SQL, al abrir una base de datos debemos indicar su nombre y la versión concreta. Posteriormente debemos crear los *almacenes de objetos*, que es muy parecido a un archivador con índices, que nos permite encontrar de una manera muy rápida el objeto que buscamos. Una vez el almacén está listo, podemos almacenar cualquier tipo de objeto con el índice que definamos. No importa el tipo de objeto que almacenemos, ni tienen que tener las mismas propiedades.

8.3.1 CREANDO Y ABRIENDO LA BD

De forma similar a como hacíamos en Web SQL, vamos a abrir una base de datos, y trabajar directamente con el objeto que nos devuelve. De nuevo, al igual que en Web SQL, las peticiones a la base de datos se realizan de manera asíncrona.

```
window.indexedDB = window.indexedDB || window.webkitIndexedDB || window.mozInde
xedDB;

if ('webkitIndexedDB' in window) {
    window.IDBTransaction = window.webkitIDBTransaction;
    window.IDBKeyRange = window.webkitIDBKeyRange;
}

var request = indexedDB.open('videos');
request.onerror = function () {
    console.log('failed to open indexedDB');
```

```
};
request.onsuccess = function (event) {
    // handle version control
    // then create a new object store
};
```

Ahora que la base de datos esta abierta (y asumiendo que no hay errores), se ejecutará el evento `onsuccess`. Antes de poder crear almacenes de objetos, tenemos que tener en cuenta los siguiente:

- Necesitamos un *manejador* para poder realizar transacciones de inserción y obtención de datos.
- Hay que especificar la versión de la base de datos. Si no existe una versión definida, significa que la base de datos no está aún creada.

El método `onsuccess` recibe como parámetro un evento, al igual que lo hace cualquier otro método escuchador de eventos. Dentro de este objeto, encontramos una propiedad llamada `target`, y dentro de ésta otra propiedad llamada `result`, que contiene el resultado de la operación. En este caso específico, `event.target.result` contiene la base de datos que acabamos de abrir.

```
var db = null;

var request = indexedDB.open('videos');
request.onsuccess = function (event) {
    // cache a copy of the database handle for the future
    db = event.target.result;
    // handle version control
    // then create a new object store
};
request.onerror = function (event) {
    alert('Something failed: ' + event.target.message);
};
```

Es importante indicar que en `IndexedDB`, los errores que se producen escalan hasta el objeto `request`. Esto quiere decir, que si un error ocurre en cualquier petición (por ejemplo una consulta de datos), en este caso mostraría una alerta con el error.

8.3.2 CONTROL DE VERSIONES

El primer paso tras abrir la base de datos es realizar un control de versiones de la base de datos. Podemos utilizar cualquier cadena de caracteres como identificador de la versión, pero lo lógico es seguir un patrón típico de software, como '0.1', '0.2', etc. Al abrir la base de datos, debemos comprobar si la versión actual de la base de datos coincide con la última versión de la aplicación. Si son diferentes, tendremos que realizar la actualización.

```
var db = null, version = '0.1';

request.onsuccess = function (event) {
    // cache a copy of the database handle for the future
    db = event.target.result;

    // handle version control
    if (version != db.version) {
        // set the version to 0.1
        var verRequest = db.setVersion(version);
        verRequest.onsuccess = function (event) {
            // now we're ready to create the object store!
        };
        verRequest.onerror = function () {
            alert('unable to set the version :' + version);
        };
    }
};
```

8.3.3 CREAR ALMACENES DE OBJETOS

Tanto la primera vez que creamos nuestra base de datos, como a la hora de actualizarla, debemos crear los almacenes de objetos correspondientes.

```
var verRequest = db.setVersion(version);
verRequest.onsuccess = function (event) {
    var store = db.createObjectStore('blockbusters', {
        keyPath: 'title',
        autoIncrement: false
    });

    // at this point we would notify our code
    // that the object store is ready
};
```

Para este ejemplo, hemos creado un único almacén de objetos, pero lo normal es disponer de varios y que puedan relacionarse entre ellos. El método `createObjectStore` admite dos parámetros:

- `name`: indica el nombre del almacén de objetos.
- `optionalParameters`: este parámetro permite definir cual va a ser el índice de los objetos, a través del cual se van a realizar las búsquedas y que por tanto debe ser único. Si no deseamos que este índice se incremente automáticamente al añadir nuevos objetos, también lo podemos indicar aquí. Al añadir un nuevo objeto, es importante asegurarnos que disponga de la propiedad definida como índice, y que

su valor sea único.

Es posible que deseemos añadir nuevos índices a nuestros objetos, con el fin de poder realizar búsquedas posteriormente. Podemos realizarlo de la siguiente manera:

```
store.createIndex('director', 'director', { unique: false });
```

Hemos añadido un nuevo índice al almacén, llamado `director` (primer argumento), y hemos indicado que el nombre de la propiedad del objeto es `director` (segundo argumento), a través del cual vamos a realizar las búsquedas. Evidentemente, varias películas pueden tener el mismo director, por lo que este valor no puede ser único. De esta manera, podemos almacenar objetos de este tipo:

```
{
    title: "Belly Dance Bruce - Final Strike",
    date: (new Date).getTime(), // released TODAY!
    director: "Bruce Awesome",
    length: 169, // in minutes
    rating: 10,
    cover: "/images/wobble.jpg"
}
```

8.3.4 AÑADIR OBJETOS AL ALMACÉN

Disponemos de dos métodos para añadir objetos al almacén: `add` y `put`.

- `add`: añade un **nuevo** objeto al almacén. Es obligatorio que los nuevos datos no existan en el almacén, de otro modo esto provocaría un error de tipo `ConstrainError`.
- `put`: en cambio, éste método actualiza el valor del objeto si existe, o lo añade si no existe en el almacén.

```
var video = {
    title: "Belly Dance Bruce - Final Strike",
    date: (new Date).getTime(), // released TODAY!
    director: "Bruce Awesome",
    length: 169, // in minutes
    rating: 10,
    cover: "/images/wobble.jpg"
};

var myIDBTransaction =
    window.IDBTransaction
    || window.webkitIDBTransaction
    || { READ_WRITE: 'readwrite' };
```

```
var transaction =
    db.transaction(['blockbusters'], myIDBTransaction.READ_WRITE);
var store = transaction.objectStore('blockbusters');
// var request = store.add(video);
var request = store.put(video);
```

Analizemos las tres últimas líneas, utilizadas para añadir un nuevo objeto al almacén:

1. `transaction = db.transaction(['blockbusters'], READ_WRITE)`: creamos una nueva transacción de lectura/escritura, sobre los almacenes indicados (en este caso sólo 'blockbusters'), pero podrían ser varios si es necesario. Si no necesitamos que la transacción sea de escritura, podemos indicarlo con la propiedad `IDBTransaction.READ_ONLY.`.
2. `store = transaction.objectStore('blockbusters')`: obtenemos el almacén de objetos sobre el que queremos realizar las operaciones, que debe ser uno de los indicados en la transacción. Con la referencia a este objeto, podemos ejecutar las operaciones de `add`, `put`, `get`, `delete`, etc.
3. `request = store.put(video)`: insertamos el objeto en el almacén. Si la transacción se ha realizado correctamente, se llamará al evento `onsuccess`, mientras que si ha ocurrido un error, se llamará a `onerror`.

8.3.5 OBTENER OBJETOS DEL ALMACÉN

El proceso para acceder a los objetos almacenados es muy similar a lo realizado para insertarlos. Seguimos necesitando una transacción, pero en este caso de sólo lectura. El proceso es el siguiente:

```
var myIDBTransaction =
    window.IDBTransaction
    || window.webkitIDBTransaction
    || { READ: 'read' };

var key = "Belly Dance Bruce - Final Strike";

var transaction =
    db.transaction(['blockbusters'], myIDBTransaction.READ);
var store = transaction.objectStore('blockbusters');
// var request = store.add(video);
var request = store.get(key);
```

En este caso, lo importante es que la clave (la variable `key`) que hemos pasado al método `get`, buscará el valor que contiene en la propiedad que hemos definido como `keyPath` al crear el almacén de objetos.

> ** *Nota* **

> *El método `get` produce el mismo resultado tanto si el objeto existe en el almacén como si no, pero con un valor `undefined`. Para evitar esta situación, es recomendable utilizar el método `openCursor()` con la misma clave. Si el objeto no existe, el valor del resultado es `null`.*

Para obtener todos los objetos de un almacén, en lugar de un único objeto, debemos hacer uso del método `openCursor`. Este método acepta como parámetro un objeto de tipo IDBKeyRange, que podemos utilizar para acotar la búsqueda.

```
var transaction =
    db.transaction(['blockbusters'], myIDBTransaction.READ);
var store = transaction.objectStore('blockbusters');
var data = [];

var request = store.openCursor();
request.onsuccess = function (event) {
    var cursor = event.target.result;
    if (cursor) {
        // value is the stored object
        data.push(cursor.value);
        // get the next object
        cursor.continue();
    } else {
        // we've got all the data now, call
        // a success callback and pass the
        // data object in.
    }
};
```

En ese ejemplo, abrimos el almacén de objetos al igual que lo hemos venido haciendo, pero en lugar de obtener un único objeto con el método `get`, abrimos un *cursor*. Esto nos permite iterar por los objetos devueltos por el cursor, todos como es este caso, o los que cumplan una condición concreta. El objeto en concreto al que apunta el cursor en la iteración actual se encuentra almacenado en su propiedad `cursor.value`. Avanzar en el cursor, para obtener el siguiente objeto, es tan sencillo como llamar al método `continue()` del cursor, lo que provocará una nueva llamada al evento `onsuccess`, siendo nosotros los que tenemos que controlar si el cursor ya no apunta a ningún objeto (`cursor === false`).

8.3.6 ELIMINAR OBJETOS DEL ALMACÉN

La última operación a realizar sobre el almacén de objetos, es eliminar datos existentes. De nuevo, el proceso es prácticamente el mismo que para obtener datos:

```
var myIDBTransaction =
    window.IDBTransaction
```

```
    || window.webkitIDBTransaction
    || { READ_WRITE: 'readwrite' };

var transaction =
    db.transaction(['blockbusters'], myIDBTransaction.READ_WRITE);
var store = transaction.objectStore('blockbusters');
var request = store.delete(key);
```

Si queremos eliminar todos los objetos de un almacén, podemos utilizar el método `clear()`, siguiente el mismo proceso visto anteriormente.

```
var request = store.clear();
```

Ejercicio 10

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 9 SIN CONEXIÓN

Si bien todos los navegadores tienen mecanismos de almacenamiento en caché, estos sistemas no son fiables y no siempre funcionan como debieran. HTML5 permite resolver algunas de las molestias asociadas al trabajo sin conexión mediante la interfaz `ApplicationCache`.

Algunas de las ventajas que conlleva el uso de ésta caché para una aplicación son:

- Navegación sin conexión: los usuarios pueden explorar todo el sitio web sin conexión.
- Velocidad: los recursos almacenados en caché son locales y, por tanto, se cargan más rápido.
- Reducción de carga del servidor: el navegador solo descarga recursos del servidor que han cambiado.

Para poder trabajar sin conexión, una aplicación únicamente necesita de un archivo de *manifiesto*, el cual indica al navegador que ficheros debe almacenar en la caché local. El contenido del manifiesto puede ser tan simple como un listado de archivos. Una vez que el navegador ha descargado y almacenado los ficheros (html, CSS, imágenes, javascripts, etc), el navegador hace uso de estos ficheros, incluso cuando el usuario actualiza la página en su navegador.

Además de especificar qué ficheros van a ser almacenados en la caché, es posible indicar cuáles no tienen que serlo, y por tanto obligar al navegador a realizar una petición de dichos ficheros al servidor. Finalmente, si intentamos acceder a un fichero no almacenado en local, y no disponemos de conexión, podemos mostrar un recurso que previamente hemos almacenado en la caché.

9.1 EL ARCHIVO DE MANIFIESTO DE CACHÉ

El archivo de manifiesto es lo que le indica al navegador cuando y qué tiene que almacenar en su caché, y qué tiene que traerse de la Web. Indicar al navegador el manifiesto que tiene que utilizar es muy sencillo:

```
<!DOCTYPE html>
<html lang="en" manifest="/example.appcache">
    ...
</html>
```

El atributo `manifest` debe estar incluido en todas las páginas de nuestra aplicación, que queramos que se almacenen en la caché. Es decir, además de los ficheros indicados en el manifiesto, la propia página que incluye el manifiesto es almacenada en la caché. El navegador no almacenará en caché ninguna página que no contenga el atributo `manifest` (a menos que esa página aparezca explícitamente en el propio archivo de manifiesto).

El atributo `manifest` puede señalar a una URL absoluta o a una ruta relativa, pero las URL absolutas deben tener el mismo origen que la aplicación web. Un archivo de manifiesto puede tener cualquier extensión, pero se debe mostrar con el tipo MIME correcto:

```
<html manifest="http://www.example.com/example.mf">
    ...
</html>
```

El tipo MIME con el que se deben mostrar los archivos de manifiesto es `text/cache-manifest`. Es posible que se tenga que añadir un tipo de archivo personalizado a la configuración de `.htaccess` o de tu servidor web.

9.1.1 ESTRUCTURA

Ejemplo de un archivo de manifiesto sencillo:

```
CACHE MANIFEST
index.html
stylesheet.css
images/logo.png
scripts/main.js
```

El archivo de manifiesto del ejemplo permite almacenar en caché los cuatro archivos especificados. El formato del manifiesto es importante:

- La cadena `CACHE MANIFEST` debe aparecer en la primera línea y es obligatoria.
- Dentro del manifiesto, los ficheros son listados dentro de categorías, también conocidos como `namespaces`. Si no se especifica ninguna categoría, todos los ficheros pertenecen a la categoría `CACHE`.

Un ejemplo más complejo sería:

```
CACHE MANIFEST
# 2010-06-18:v2

CACHE:
/favicon.ico
index.html
```

```
stylesheet.css
images/logo.png
scripts/main.js

# Resources that require the user to be online.
NETWORK:
login.php
/myapi
http://api.twitter.com

# static.html will be served if main.py is inaccessible
# offline.jpg will be served in place of all images in images/large/
# offline.html will be served in place of all other .html files
FALLBACK:
/main.py /static.html
images/large/ images/offline.jpg
*.html /offline.html
```

Un archivo de manifiesto puede incluir tres categorías: `CACHE`, `NETWORK` y `FALLBACK`.

- `CACHE`: esta es la sección predeterminada para las entradas. Los archivos incluidos en esta sección (o inmediatamente después de `CACHE MANIFEST`) se almacenarán en caché explícitamente después de descargarse por primera vez.
- `NETWORK`: los archivos incluidos en esta sección son recursos permitidos que requieren conexión al servidor. En todas las solicitudes enviadas a estos recursos se omite la caché, incluso si el usuario está trabajando sin conexión. Se pueden utilizar caracteres comodín.
- `FALLBACK`: se trata de una sección opcional en la que se especifican páginas alternativas en caso de no poder acceder a un recurso. La primera `URI` corresponde al recurso y la segunda, a la página alternativa. Ambas `URI` deben estar relacionadas y tener el mismo origen que el archivo de manifiesto. Se pueden utilizar caracteres comodín.

```
CACHE MANIFEST
# 2010-06-18:v3

# Explicitly cached entries
index.html
css/style.css

# offline.html will be displayed if the user is offline
FALLBACK:
/ /offline.html

# All other resources require the user to be online.
```

```
NETWORK:
*

# Additional resources to cache
CACHE:
images/logo1.png
images/logo2.png
images/logo3.png
```

> ***Nota***
>
> *Las peticiones de recursos que den como resultado un error 404 (por ejemplo una imagen no encontrada), mostrarán en este caso el fichero `offline.html`*

9.2 CÓMO SERVIR EL MANIFIESTO

Como hemos comentado anteriormente, el manifiesto puede tener cualquier extensión (aunque se recomienda que sea .appcache), pero lo importante es que el servidor envíe el fichero con el tipo MIME correcto. Si utilizamos *Apache*, es tan sencillo como añadir la siguiente línea al fichero `mime.types`:

```
text/cache-manifest appcache
```

Esta configuración dependerá del servidor web que utilicemos. De todas maneras, para asegurarnos que el servidor está enviando el manifiesto con la cabecera correcta, podemos utilizar una herramienta como `curl` de la siguiente manera:

```
curl -I http://mysite.com/manifest.appcache
```

O bien a través de las herramientas de desarrollo integradas en Google Chrome, Safari y Firefox. De cualquiera de las maneras, la respuesta tendría que ser algo parecido a esto:

```
HTTP/1.1 200 OK
Date: Mon, 13 Sep 2010 12:59:30 GMT
Server: Apache/2.2.13 (Unix) mod_ssl/2.2.13 OpenSSL/0.9.8l DAV/2 PHP/5.3.0
Last-Modified: Tue, 31 Aug 2010 03:11:00 GMT
Accept-Ranges: bytes
Content-Length: 113

Content-Type: text/cache-manifest
```

9.3 PROCESO DE CACHEADO

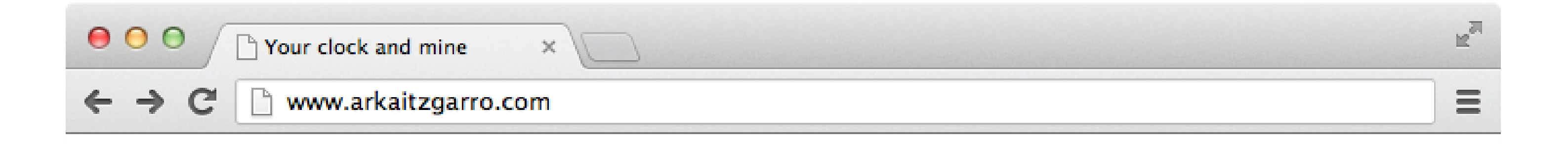
Cuando visitamos una página que hace uso de la cache de aplicación, el proceso de cacheado que se sigue es el siguiente:

1. **Navegador:** solicita la página http://html5app.com/
2. **Servidor:** devuelve index.html
3. **Navegador:** procesa la página index.html y solicita los recursos asociados, como imágenes, javascripts, hojas de estilos y el manifiesto.
4. **Servidor:** devuelve todos los recursos solicitados.
5. **Navegador:** procesa el manifiesto y solicita, de nuevo, todos los recursos definidos en el manifiesto. Efectivamente, se produce una doble petición.
6. **Servidor:** devuelve todos los recursos del manifiesto solicitados.
7. **Navegador:** la cache de aplicación está actualizada, y se lanzan los eventos asociados.

Ahora, el navegador está listo y la caché contiene los ficheros indicados en el manifiesto. Si el manifiesto no ha cambiado, y la página se recarga, ocurre lo siguiente:

1. **Navegador:** vuelve a solicitar la página http://html5app.com/
2. **Navegador:** detecta que tiene una copia local de index.html y la sirve de manera local.
3. **Navegador:** procesa la página index.html y los recursos existentes en la caché se sirven de manera local.
4. **Navegador:** solicita de nuevo el manifiesto al servidor.
5. **Servidor:** devuelve un código 304 indicando que no ha cambiado nada en el manifiesto.

Una vez que el navegador tiene almacenados los recursos en su caché, los sirve de manera local y después solicita el manifiesto. Como se puede ver en la siguiente captura de pantalla, Google Chrome únicamente solicita al servidor aquellos ficheros que no se encuentran en la caché de la aplicación.



The time on your computer is **22:17:12** and the time on the server is **10:30:10**

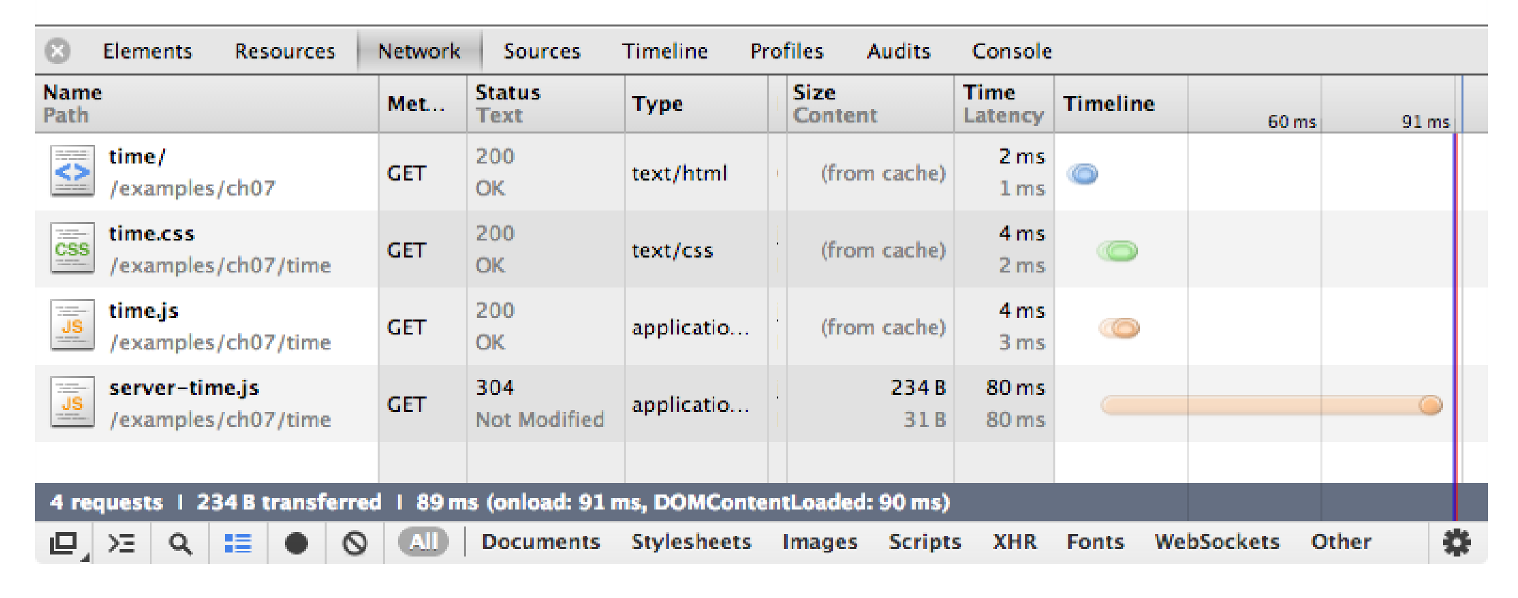


Figura 9.1 Peticiones realizadas por Google Chrome

Si deseamos actualizar alguno de los recursos de la aplicación, tendremos que actualizar primero el manifiesto, para obligar al navegador a solicitar de nuevo todos los recursos. Para ello, es necesario *marcar* de alguna manera que el manifiesto ha cambiado, aunque los ficheros a cachear sean los mismos. Una práctica muy sencilla es añadir una número de versión o la fecha de modificación del manifiesto:

```
# 2010-06-18:v3
```

Una vez que el manifiesto ha cambiado, el comportamiento del navegador es el siguiente:

1. **Navegador:** vuelve a solicitar la página http://html5app.com/
2. **Navegador:** detecta que tiene una copia local de index.html y la sirve de manera local.
3. **Navegador:** procesa la página index.html **y los recursos existentes en la caché se sirven de manera local**.
4. **Navegador:** solicita de nuevo el manifiesto al servidor.
5. **Servidor:** devuelve el nuevo manifiesto modificado.
6. **Navegador:** procesa el manifiesto y solicita todos los recursos definidos en el manifiesto.
7. **Servidor:** devuelve todos los recursos del manifiesto solicitados.

8. **Navegador:** la cache de aplicación está actualizada, y se lanzan los eventos asociados.

Hay que destacar, que a pesar de haber modificado los recursos en el navegador, estos cambios no se producen en este momento, ya que se siguen utilizando los cargados previamente. La nueva caché solo estaría disponible si volviésemos a recargar la página. Una manera de modificar este comportamiento es accediendo al objeto `applicationCache`.

9.4 ACTUALIZACIÓN DE LA MEMORIA CACHÉ

El objeto `window.applicationCache` permite acceder mediante JavaScript a la caché de aplicación del navegador. Su propiedad `status` permite comprobar el estado de la memoria caché, y es el encargado de notificarnos que se ha producido un cambio en la caché local.

```
var appCache = window.applicationCache;
switch (appCache.status) {
    case appCache.UNCACHED: // UNCACHED == 0
        return 'UNCACHED'; break;
    case appCache.IDLE: // IDLE == 1
        return 'IDLE'; break;
    case appCache.CHECKING: // CHECKING == 2
        return 'CHECKING'; break;
    case appCache.DOWNLOADING: // DOWNLOADING == 3
        return 'DOWNLOADING'; break;
    case appCache.UPDATEREADY:  // UPDATEREADY == 4
        return 'UPDATEREADY'; break;
    case appCache.OBSOLETE: // OBSOLETE == 5
        return 'OBSOLETE'; break;
    default:
        return 'UKNOWN CACHE STATUS'; break;
};
```

Para actualizar la caché mediante JavaScript, primero se debe hacer una llamada a `applicationCache.update()`. Al hacer esa llamada, se intentará actualizar la caché del usuario (para lo cual será necesario que haya cambiado el archivo de manifiesto). Finalmente, cuando el estado de `applicationCache.status` sea `UPDATEREADY`, al llamar a `applicationCache.swapCache()`, se sustituirá la antigua caché por la nueva.

```
var appCache = window.applicationCache;

appCache.update(); // Attempt to update the user's cache.

if (appCache.status == window.applicationCache.UPDATEREADY) {
    appCache.swapCache();  // The fetch was successful, swap in the new cache.
```

```
}
```

Al utilizar `update()` y `swapCache()` de este modo, no se muestran los recursos actualizados a los usuarios. El flujo indicado solo sirve para pedirle al navegador que busque un nuevo archivo de manifiesto, que descargue el contenido actualizado que se especifica y que actualice la caché de la aplicación. Por tanto, la página se tiene que volver a cargar dos veces para que se muestre el nuevo contenido a los usuarios: una vez para extraer una nueva caché de aplicación y otra para actualizar el contenido de la página.

Para que los usuarios puedan acceder a la versión más reciente del contenido de tu sitio, podemos establecer un escuchador que controle el evento `updateready` cuando se cargue la página:

```
window.addEventListener('load', function(e) {
    window.applicationCache.addEventListener('updateready', function(e) {
        if (window.applicationCache.status == window.applicationCache.UPDATEREA
DY) {
            // Browser downloaded a new app cache.
            // Swap it in and reload the page to get the new hotness.
            window.applicationCache.swapCache();
            if (confirm('A new version of this site is available. Load it?')) {
                window.location.reload();
            }
        } else {
            // Manifest didn't changed. Nothing new to server.
        }
    }, false);
}, false);
```

Hay, además, algunos eventos adicionales que permiten controlar el estado de la caché. El navegador activa eventos para una serie de acciones (como el progreso de las descargas, la actualización de la caché de las aplicaciones y los estados de error). El siguiente fragmento permite establecer escuchadores de eventos para cada tipo de evento de caché:

```
function handleCacheEvent(e) {}

function handleCacheError(e) {
  alert('Error: Cache failed to update!');
};

// Fired after the first cache of the manifest.
appCache.addEventListener('cached', handleCacheEvent, false);

// Checking for an update. Always the first event fired in the sequence.
```

```
appCache.addEventListener('checking', handleCacheEvent, false);

// An update was found. The browser is fetching resources.
appCache.addEventListener('downloading', handleCacheEvent, false);

// The manifest returns 404 or 410, the download failed,
// or the manifest changed while the download was in progress.
appCache.addEventListener('error', handleCacheError, false);

// Fired after the first download of the manifest.
appCache.addEventListener('noupdate', handleCacheEvent, false);

// Fired if the manifest file returns a 404 or 410.
// This results in the application cache being deleted.
appCache.addEventListener('obsolete', handleCacheEvent, false);

// Fired for each resource listed in the manifest as it is being fetched.
appCache.addEventListener('progress', handleCacheEvent, false);

// Fired when the manifest resources have been newly redownloaded.
appCache.addEventListener('updateready', handleCacheEvent, false);
```

Si no se puede descargar el archivo de manifiesto o algún recurso especificado en él, fallará todo el proceso de actualización. Si se produce ese fallo, el navegador seguirá utilizando la antigua caché de la aplicación.

9.5 EVENTOS ONLINE/OFFLINE

Como parte de la especificación de HTML5, el objeto `navigator` incluye una propiedad que nos indica si se dispone de conexión o no, concretamente `navigator.onLine`. Sin embargo, esta propiedad no se comporta de manera correcta en la mayoría de navegadores, y únicamente cambia su estado al indicar de manera explícita que funcione en *modo offline*. Como desarrolladores, lo que realmente nos interesa es conocer si realmente hay conexión o no con el servidor.

Una manera de identificar si existe conexión a internet, es utilizar la categoría FALLBACK del manifiesto. En esta categoría podemos indicar dos ficheros JavaScript que detectan si estamos online o no:

```
CACHE MANIFEST

FALLBACK:
online.js offline.js
```

`online.js` contiene:

```
setOnline(true);
```

Y `offline.js` contiene:

```
setOnline(false);
```

En nuestra aplicación, creamos una función llamada `testOnline` que dinámicamente crea un elemento `<script>`, el cual trata de cargar el fichero `online.js`. Si la carga se realiza de manera correcta, se ejecuta el método `setOnline(true)`. Si estamos *offline*, el navegador cargará el fichero `offline.js`, ejecutando el método `setOnline(false)`.

```
function testOnline(fn) {
    var script = document.createElement(‘script’)
    script.src = 'online.js';
    // alias the setOnline function to the new function that was passed in
    window.setOnline = function (online) {
        document.body.removeChild(script);
        fn(online);
    };

    // attaching script node trigger the code to run
    document.body.appendChild(script);
}

testOnline(function (online) {
    if (online) {
        applicationCache.update();
    } else {
        // show users an unobtrusive message that they're disconnected
    }
});
```

Ejercicio 11

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 10 DRAG AND DROP

Durante años, hemos utilizado bibliotecas como *jQuery* y *Dojo* para conseguir funcionalidades complejas en las *interfaces* de usuario como las animaciones, las esquinas redondeadas y la función de arrastrar y soltar. Esta última funcionalidad (*Drag and Drop, DnD*) tiene una gran importancia en HTML5, y de hecho se ha integrado en el API. En la especificación, este API se define como un mecanismo basado en eventos, donde identificamos los elementos que deseamos arrastrar con el atributo `draggable` y desde JavaScript escuchamos los eventos que se producen, para proporcionar la funcionalidad deseada.

Por defecto, todos los enlaces, imágenes y nodos de texto (o selecciones de texto) de un documento HTML son arrastables, pero no hay ningún evento asociado a estas acciones, por lo que poco más podemos hacer, a excepción de la funcionalidad que nos ofrezca el navegador o el propio sistema operativo (guardar las imágenes en el escritorio, crear ficheros de texto, etc).

10.1 DETECCIÓN DE LA FUNCIONALIDAD

Aunque la compatibilidad actual de los navegadores con esta API es bastante amplia, hay que tener en cuenta que los dispositivos móviles no soportan esta funcionalidad. Si nuestro sitio web está siendo accedido desde un dispositivo móvil, y tenemos implementada esta funcionalidad (por ejemplo en una cesta de la compra), debemos proveer otra solución para que nuestro sitio web se comporte de manera correcta, y no perjudicar la experiencia del usuario.

La manera mas sencilla de comprobar la disponibilidad de este API, es utilizar la biblioteca *Modernizr*, la cual nos indica si el navegador soporta esta funcionalidad:

```
if (Modernizr.draganddrop) {
    // Browser supports HTML5 DnD.
} else {
    // Fallback to a library solution or disable DnD.
}
```

10.2 CREACIÓN DE CONTENIDO *ARRASTRABLE*

Hacer que un elemento se pueda arrastrar es muy sencillo. Solo hay que establecer el atributo `draggable="true"` en el elemento que se quiere mover. La función de arrastre se puede habilitar prácticamente en cualquier elemento, incluidos archivos, imágenes,

enlaces, listas u otros nodos DOM.

```
<div id="columns">
    <div class="column" draggable="true"><header>A</header></div>
    <div class="column" draggable="true"><header>B</header></div>
    <div class="column" draggable="true"><header>C</header></div>
</div>
```

Una *ayuda visual* al usuario, para indicar que un elemento es arrastable, es transformar el aspecto tanto del elemento como del cursor. Con CSS esto es muy sencillo:

```
[draggable] {
    user-select: none;
}

.column:hover {
    border: 2px dotted #666666;
    background-color: #ccc;
    border-radius: 10px;
    box-shadow: inset 0 0 3px #000;
    cursor: move;
}
```

10.3 EVENTOS DE ARRASTRE

La especificación define hasta siete eventos que son lanzados tanto por los elementos de origen (los que son arrastrados) com para los elementos de destino (donde *soltamos* el elemento arrastrado). Son los siguientes:

- `dragstart`: comienza el arrastrado. El `target` del evento hace referencia al elemento que está siendo arrastrado.
- `drag`: el elemento se ha movido. El `target` del evento hace referencia al elemento que está siendo arrastrado. Este evento se dispara tantas veces como se mueva el elemento.
- `dragenter`: se dispara cuando un elemento que está siendo arrastrado entra en un contenedor. El `target` del evento hace referencia al elemento contenedor.
- `dragleave`: el elemento arrastrado ha salido del contenedor. El `target` del evento hace referencia al elemento contenedor.
- `dragover`: el elemento se ha movido dentro del contenedor. El `target` del evento hace referencia al elemento contenedor. Como el comportamiento por defecto es denegar el `drop`, la función debe retornar el valor `false` o llamar al método `preventDefault` del evento para que indicar que se puede el soltar elemento.
- `drop`: el elemento arrastrado has sido exitosamente soltado en el elemento

contenedor. El `target` del evento hace referencia al elemento contenedor.

- `dragend`: se ha dejado de arrastrar el elemento, con éxito o no. El `target` del evento hace referencia al elemento arrastrado.

Para organizar el flujo de *DnD*, necesitamos un elemento de origen (en el que se origina el movimiento de arrastre), la carga de datos (la información que va asociada al elemento arrastrado) y un elemento de destino (el área en la que se soltarán los datos). El elemento de origen puede ser una imagen, una lista, un enlace, un objeto de archivo, un bloque de HTML o cualquier otro elemento, al igual que la zona de destino.

10.3.1 COMIENZO DE LA OPERACIÓN DE ARRASTRE

Una vez que se haya definido el atributo `draggable="true"` en los elementos que queremos convertir en arrastables, debemos añadir los escuchadores necesarios para reaccionar antes los eventos que se lanzan desde los elementos. El primer evento de los que define la especificación se produce cuando un elemento comienza a ser arrastrado. Un ejemplo muy sencillo en el que cambiamos la transparencia de un elemento al comenzar a ser arrastrado:

```
function handleDragStart(e) {
    this.style.opacity = '0.4';
}

var cols = document.querySelectorAll('#columns .column');
[].forEach.call(cols, function(col) {
    col.addEventListener('dragstart', handleDragStart, false);
});
```

No debemos olvidarnos de volver a fijar la transparencia del elemento en el 100% una vez completada la operación de arrastre, ya que de otro modo, seguiría siendo transparente una vez finalizado la operación de arrastre.

10.3.2 `DRAGENTER`, `DRAGOVER` Y `DRAGLEAVE`

Los controladores de eventos `dragenter`, `dragover` y `dragleave` se pueden utilizar para proporcionar pistas visuales adicionales durante el proceso de arrastre. Por ejemplo, el borde de un elemento de destino se puede convertir en una línea discontinua al realizar una operación de arrastre. De esa forma, los usuarios podrán distinguir cuáles son los elementos de destino donde pueden soltar lo que están arrastrando.

```
.column.over {
    border: 2px dashed #000;
}
```

```
function handleDragStart(e) {
    this.style.opacity = '0.4';
```

```
}

function handleDragOver(e) {
    if (e.preventDefault) {
        e.preventDefault(); // Necessary. Allows us to drop.
    }
    e.dataTransfer.dropEffect = 'move';
    return false;
}

function handleDragEnter(e) {
    // this / e.target is the current hover target.
    this.classList.add('over');
}

function handleDragLeave(e) {
    this.classList.remove('over');  // this / e.target is previous target eleme
nt.
}

var cols = document.querySelectorAll('#columns .column');
[].forEach.call(cols, function(col) {
    col.addEventListener('dragstart', handleDragStart, false);
    col.addEventListener('dragenter', handleDragEnter, false);
    col.addEventListener('dragover', handleDragOver, false);
    col.addEventListener('dragleave', handleDragLeave, false);
});
```

Algunas consideraciones en el código anterior. Cuando *algo* es soltado sobre un elemento, se dispara el evento `dragover`, y tenemos la posibilidad de leer el objeto `event.dataTransfer`. Este objeto contiene los datos asociados a las operaciones de arrastre, como veremos más adelante. Tal y como hemos comentado anteriormente, cuando soltamos un elemento, hay que impedir el comportamiento predeterminado del navegador, que es denegar el `drop`.

10.3.3 FINALIZACIÓN DE LA OPERACIÓN DE ARRASTRE

Para que se procese la operación de soltar, se debe añadir un escuchador para los eventos `drop` y `dragend`. En este controlador, habrá que impedir el comportamiento predeterminado del navegador para este tipo de operaciones, que puede ser abrir un enlace o mostrar una imagen. Para ello, evitamos la propagación del evento con `e.stopPropagation()`.

En nuestro ejemplo utilizaremos `dragend` para eliminar la clase `over` de cada columna:

```
function handleDrop(e) {
```

```
    // this / e.target is current target element.
    if (e.stopPropagation) {
        e.stopPropagation(); // stops the browser from redirecting.
    }
    // See the section on the DataTransfer object.
    return false;
}

function handleDragEnd(e) {
    // this/e.target is the source node.
    [].forEach.call(cols, function (col) {
        col.classList.remove('over');
    });
}

var cols = document.querySelectorAll('#columns .column');
[].forEach.call(cols, function(col) {
    col.addEventListener('dragstart', handleDragStart, false);
    col.addEventListener('dragenter', handleDragEnter, false);
    col.addEventListener('dragover', handleDragOver, false);
    col.addEventListener('dragleave', handleDragLeave, false);
    col.addEventListener('drop', handleDrop, false);
    col.addEventListener('dragend', handleDragEnd, false);
});
```

10.3.4 EL OBJETO DATATRANSFER

La propiedad `dataTransfer` es el centro de desarrollo de toda la actividad de la función *DnD*, ya que contiene los datos que se envían en la acción de arrastre. La propiedad `dataTransfer` se establece en el evento `dragstart` y se lee/procesa en el evento `drop`. Al activar `e.dataTransfer.setData(format, data)`, se establece el contenido del objeto en el tipo MIME y se transmite la carga de datos en forma de argumentos.

En nuestro ejemplo, la carga de datos se ha establecido en el propio HTML del elemento de origen:

```
var dragSrcEl = null;
function handleDragStart(e) {
    this.style.opacity = '0.4';
    dragSrcEl = this;
    e.dataTransfer.effectAllowed = 'move';
    e.dataTransfer.setData('text/html', this.innerHTML);
}
```

`dataTransfer` también tiene el formato `getData` necesario para la extracción de los datos de arrastre por tipo de MIME. A continuación se indica la modificación necesaria para el procesamiento de la acción de arrastre de un elemento:

```
function handleDrop(e) {
    // this/e.target is current target element.
    if (e.stopPropagation) {
            e.stopPropagation(); // Stops some browsers from redirecting.
    }
    // Don't do anything if dropping the same column we're dragging.
    if (dragSrcEl != this) {
            // Set the source column's HTML to the HTML of the columnwe dropped
 on.
            dragSrcEl.innerHTML = this.innerHTML;
            this.innerHTML = e.dataTransfer.getData('text/html');
    }
    return false;
}
```

Hemos añadido una variable global llamada `dragSrcEl` para facilitar el cambio de posición de la columna. En `handleDragStart()`, la propiedad `innerHTML` de la columna de origen se almacena en esa variable y, posteriormente, se lee en `handleDrop()` para cambiar el HTML de las columnas de origen y destino.

10.4 ARRASTRE DE ARCHIVOS

Las API de *DnD* permiten arrastrar archivos del escritorio a una aplicación web en la ventana del navegador.

Para arrastrar un archivo desde el escritorio, se deben utilizar los eventos de *DnD* del mismo modo que otros tipos de contenido. La diferencia principal se encuentra en el controlador drop. En lugar de utilizar `dataTransfer.getData()` para acceder a los archivos, sus datos se encuentran en la propiedad `dataTransfer.files`:

```
function handleDrop(e) {
    e.stopPropagation();
    // Stops some browsers from redirecting.
    e.preventDefault();
    var files = e.dataTransfer.files;
    for (var i = 0, f; f = files[i]; i++) {
        // Read the File objects in this FileList.
    }
}
```

Ejercicio 12

Ver enunciado

ÍNDICE DE CONTENIDOS

Drag and Drop

CAPÍTULO 11 GEOLOCALIZACIÓN

El uso del API de geolocalización es extremadamente sencillo. Soportado por todos los navegadores modernos, nos permite conocer la posición del usuario con mayor o menor precisión, según el método de localización utilizado. En la actualidad, disponemos de tres *tecnologías* para geolocalizar un usuario:

- Vía IP: todo dispositivo que se encuentra conectado a la red, tiene asignada una dirección IP (*Internet Protocol*) pública que actúa, de forma muy simplificada, como un código postal. Evidentemente, esta no es la mejor manera de localización, pero sí nos da una ligera idea de dónde se encuentra.
- Redes GSM: cualquier dispositivo que se conecte a una red telefonía, es capaz de obtener una posición aproximada basándose en una triangulación con las antenas de telefonía. Es un método sensiblemente más preciso que mediante la dirección IP, pero mucho menos que mediante GPS.
- GPS: *Global Positioning System* o Sistema de Posicionamiento Global. Es el método más preciso, pudiendo concretar la posición del usuario con un margen de error de escasos metros.

El primer paso es comprobar es la disponibilidad del API de geolocalización de HTML 5 en el explorador del usuario:

```
if(Modernizr.geolocation) {
    alert('El explorador soporta geolocalización');
} else {
    alert('El explorador NO soporta geolocalización');
}
```

El API ofrece los siguientes métodos para geolocalizar la posición del usuario:

- `getCurrentPosition`: obtiene la posición actual del usuario, utilizando la mejor tecnología posible.
- `watchPosition`: consulta cada cierto tiempo la posición del usuario, ejecutando la función de *callback* indicada únicamente si la posición ha cambiado desde la última consulta.

Ambos métodos se ejecutan de manera asíncrona para obtener la posición del usuario. Si es la primera vez que se solicita la localización al navegador, éste mostrará un mensaje pidiendo permiso al usuario para compartir su localización. Si el usuario no da su

permiso, el API llama a la función de error que hayamos definido. La especificación dice:

> *"El navegador no debe enviar información sobre la localización a sitios sin el permiso explícito del usuario."*

Así pues, queda en manos de los navegadores informar al usuario que estamos intentando acceder a su posición actual. La forma de realizarlo depende del navegador. Por norma general, los navegadores de escritorio muestran un aviso no bloqueante, lo que permite seguir utilizando y ejecutando la aplicación.



Figura 11.1 Petición para compartir localización en Chrome

En cambio, en navegadores de dispositivos móviles, como Safari y Chrome, se muestra una ventana modal que bloquea la ejecución del código hasta que el usuario acepte o deniegue la solicitud de geolocalización.



Figura 11.2 Petición para compartir localización en Safari Mobile y Chrome Mobile

11.1 MÉTODOS DEL API

El API de geolocalización del objeto `navigator` contiene tres métodos:

- `getCurrentPosition`
- `watchPosition`
- `clearWatch`

Los métodos `watchPosition` y `clearWatch` están emparejados, de la misma manera que lo están `setInterval` y `clearTimeout`. `watchPosition` devuelve un identificador único, que permite cancelar posteriormente las consultas de posición pasando es identificador como parámetro a `clearWatch`.

Tanto `getCurrentPosition` como `watchPosition`, son métodos muy parecidos, y toman los mismos parámetros: una función de éxito, una función de error y opciones de geolocalización. Un simple ejemplo de uso del API geolocalización es el siguiente:

```
navigator.geolocation.getCurrentPosition(function (position) {
    alert('We found you!');
    // now do something with the position data
});
```

11.1.1 FUNCIÓN DE ÉXITO

Si el usuario permite que el navegador comparta la localización, y no se produce ningún otro error, se llama a la función de éxito, que es el primer argumento de las funciones de `getCurrentPosition` y `watchPosition`. Ésta función recibe como parámetro un objeto `position` que contiene dos propiedades: un objeto `coords` (contiene las coordenadas) una marca de tiempo `timestamp`. El objeto de coordenadas es el que nos interesa, ya que es el que contiene la información sobre la geolocalización. Sus propiedades son las siguientes:

- `readonly attribute double latitude`
- `readonly attribute double longitude`
- `readonly attribute double accuracy`

La propiedad `accuracy` contiene la precisión de las coordenadas en metros. Podemos utilizarlo para mostrar un radio de precisión de la posición en nuestro mapa.

Aunque es complicado de confirmar, es posible que esta información de geolocalización provenga de servicios propios de los fabricantes. Por ejemplo, Google dispone de una gran base de datos, que combinado con la información de la petición (como el hardware, la dirección IP, etc.) puede dar una localización. Esta información simplemente representa la posición del usuario, y no contiene nada más que nos pueda indicar la velocidad o dirección del usuario.

Utilizando la información de posición, es muy sencillo mostrar la posición del usuario en un mapa:

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(function (position) {
        var coords = position.coords;
        showMap(coords.latitude, coords.longitude, coords.accuracy);
    });
}
```

Existe otro tipo de datos dentro del objeto `coords`, para dispositivos que dispongan de GPS, aunque la gran mayoría de navegadores establecerán estas propiedades como `null`.

- `readonly attribute double altitude`
- `readonly attribute double altitudeAccuracy`
- `readonly attribute double heading`
- `readonly attribute double speed`

11.1.2 OBTENIENDO DATOS EXTRA

Actualmente, muchos de los dispositivos integran un GPS. Dependiendo del navegador o del sistema operativo, la información provista con la geolocalización puede incluir mucha más información que simplemente las coordenadas de la posición, como la velocidad y la altitud.

En la mayoría de los casos, hay que especificar al API que utilice una mayor precisión para activar el GPS. Como siempre, hay que tener cuidado al utilizar el GPS, ya que consume muchísima batería y hay que utilizar esta tecnología únicamente si es estrictamente necesario.

Para calcular la velocidad, el dispositivo necesita conocer la diferencia media entre las últimas localizaciones. Por esta razón, es necesario utilizar el método `watchPosition` y calcular la velocidad de la siguiente manera:

```
var speedEl = document.getElementById(‘speed’);
navigator.geolocation.watchPosition(function (geodata) {
    var speed = geodata.coords.speed;
    if (speed === null || speed === 0) {
        speedEl.innerHTML = "You're standing still!";
    } else {
        // speed is in metres per second
        speedEl.innerHTML = speed + “Mps”;
    }
}, function () {
```

```
    speedEl.innerHTML = "Unable to determine speed :-(";
},
{ enableHighAccuracy: true }
);
```

11.1.3 FUNCIÓN DE ERROR

El segundo parámetro para los métodos `getCurrentPosition` y `watchPosition` es la función de error. Esta función es importante si queremos proveer de un segundo método alternativo para introducir la localización, por ejemplo de manera manual, o queremos informar al usuario del error. Esta función se ejecuta cuando el usuario deniega la petición de localización, o cuando estamos consultando la posición pero el dispositivo pierde la recepción de la localización. La función de error recibe un único argumento con dos propiedades:

- `readonly attribute unsigned short code`
- `readonly attribute DOMString message`

El código de error puede contener los siguientes valores:

- `PERMISSION_DENIED (valor = 1)`
- `POSITION_UNAVAILABLE (valor = 2)`
- `TIMEOUT (valor = 3)`

11.1.4 CONFIGURAR LA GEOLOCALIZACIÓN

Finalmente, el tercer argumento para los métodos `getCurrentPosition` y `watchPosition` contiene las opciones de geolocalización. Todas estas configuraciones son opcionales, puede que no se tengan en cuenta por los navegadores.

- `enableHighAccuracy`: booleano, por defecto `false`
- `timeout`: en milisegundos, por defecto infinito
- `maximumAge`: en milisegundos, por defecto 0

Por ejemplo, para solicitar una alta precisión, un `timeout` de dos segundos y no cachear las peticiones, llamamos al método `getCurrentPosition` con las siguientes opciones:

```
navigator.geolocation.getCurrentPosition(success, error, {
    enableHighAccuracy: true
    timeout: 2000
    maximumAge: 0
});
```

Ejercicio 13

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 12 WEB WORKERS

Los navegadores ejecutan las aplicaciones en un único *thread*, lo que significa que si JavaScript está ejecutando una tarea muy complicada, que se traduce en tiempo de procesado, el rendimiento del navegador se ve afectado. Los *Web workers* se introdujeron con la idea de simplificar la ejecución de *threads* en el navegador. Un *worker* permite crear un entorno en el que un bloque de código JavaScript puede ejecutarse de manera paralela sin afectar al *thread* principal del navegador. Los *Web workers* utilizan un protocolo de paso de mensajes similar a los utilizados en programación paralela.

Estos *Web workers* se ejecutan en un subproceso aislado. Como resultado, es necesario que el código que ejecutan se encuentre en un archivo independiente. Sin embargo, antes de hacer esto, lo primero que se tiene que hacer es crear un nuevo objeto *Worker* en la página principal:

```
var worker = new Worker('task.js');
```

Si el archivo especificado existe, el navegador generará un nuevo subproceso de *Worker* que descargará el archivo JavaScript de forma asíncrona. El *Worker* no comenzará a ejecutarse hasta que el archivo se haya descargado completamente. Si la ruta al nuevo *Worker* devuelve un `error 404`, el *Worker* fallará automáticamente.

Antes de comenzar a utilizar los *Worker*, es necesario conocer el protocolo de paso de mensajes, que también es utilizado en otras APIs como *WebSocket* y *Server-Sent Event*.

12.1 TRANSFERENCIA DE MENSAJES

El API de transferencia de mensajes es una manera muy simple de enviar cadenas de caracteres entre un origen (o un dominio) a un destino. Por ejemplo podemos utilizarlo para enviar información a una ventana abierta como *popup*, o a un *iframe* dentro de la página, aún cuando tiene como origen otro dominio.

La comunicación entre un *Worker* y su página principal se realiza mediante un modelo de evento y el método `postMessage()`. En función del navegador o de la versión, `postMessage()` puede aceptar una cadena o un objeto `JSON` como argumento único. Las últimas versiones de los navegadores modernos son compatibles con la transferencia de objetos `JSON`. De todas maneras, siempre podemos utilizar los métodos `JSON.stringify` y `JSON.parse` para la transferencia de objetos entre el *thread* principal y los *Worker*.

A continuación, se muestra un ejemplo sobre cómo utilizar una cadena para transferir "Hello World" a un *Worker* en `doWork.js`. El *Worker* simplemente devuelve el mensaje que se le transfiere.

Secuencia de comandos principal:

```
worker.postMessage('Hello World'); // Send data to our worker.
```

`doWork.js` (el Worker):

```
self.addEventListener('message', function(e) {
    self.postMessage(e.data);
}, false);
```

Cuando se ejecuta `postMessage()` desde la página principal, el *Worker* es capaz de obtener este mensaje escuchando al evento `message`. Se puede acceder a los datos del mensaje (en este caso "Hello World") a través de la propiedad `data` del evento. Aunque este ejemplo concreto no es demasiado complejo, demuestra que `postMessage()` también sirve para transferir datos de vuelta al *thread* principal, una vez que los datos de origen se hayan procesado correctamente.

Los mensajes que se transfieren entre el origen y los *Worker* se copian, no se pasan por referencia. Por ejemplo, en el siguiente ejemplo, a la propiedad `msg` del mensaje `JSON` se accede en las dos ubicaciones. Parece que el objeto se transfiere directamente al *Worker* aunque se esté ejecutando en un espacio específico e independiente. En realidad, lo que ocurre es que el objeto se serializa al transferirlo al *Worker* y, posteriormente, se anula la serialización en la otra fase del proceso. El origen y el *Worker* no comparten la misma instancia, por lo que el resultado final es la creación de un duplicado en cada transferencia. La mayoría de los navegadores implementan esta función mediante la codificación/descodificación `JSON` automática del valor en la otra fase del proceso, cuando el paso de objetos está soportado.

En el siguiente ejemplo, que es más complejo, se transfieren mensajes utilizando objetos JavaScript.

Secuencia de comandos principal:

```
<button onclick="sayHI()">Say HI</button>
<button onclick="unknownCmd()">Send unknown command</button>
<button onclick="stop()">Stop worker</button>
<output id="result"></output>
```

```
<script>
```

```
function sayHI() {
```

```
    worker.postMessage({'cmd': 'start', 'msg': 'Hi'});
}
```

```
function stop() {
    worker.postMessage({'cmd': 'stop', 'msg': 'Bye'});
}
```

```
function unknownCmd() {
    worker.postMessage({'cmd': 'foobard', 'msg': '???'});
}
```

```
var worker = new Worker('doWork.js');
```

```
worker.addEventListener('message', function(e) {
    document.getElementById('result').textContent = e.data;
}, false);
```

```
</script>
```

doWork.js:

```
this.addEventListener('message', function(e) {
    var data = e.data;
    switch (data.cmd) {
        case 'start':
            this.postMessage('WORKER STARTED: '+data.msg);
            break;
        case 'stop':
            this.postMessage('WORKER STOPPED: '+data.msg+'. (buttons will no lo
nger work)');
            this.close(); // Terminates the worker.
            break;
        default:
            this.postMessage('Unknown command: '+data.msg);
    };
}, false);
```

12.2 UTILIZACIÓN DE WEB WORKERS

Un *Worker* es una manera ejecutar código JavaScript de manera paralela al proceso principal, sin interferir con el navegador. El navegador sigue siendo responsable de solicitar y analizar ficheros, renderizar la vista, ejecutar JavaScript y cualquier otro proceso que consuma tiempo de procesado y que haga que el resto de tareas tengan que esperar. Y es aquí donde los *Web workers* toman importancia.

Al igual que con el resto de funcionalidades de HTML5, debemos comprobar su disponibilidad en el navegador en el que ejecutamos la aplicación:

```
if(Modernizr.webworkers) {
    alert('El explorador soporta Web workers');
} else {
    alert('El explorador NO soporta Web workers');
}
```

Crear nuevo *Worker* es muy sencillo. Tan sólo tenemos que crear una nueva instancia del objeto `Worker`, indicando como parámetro del constructor el fichero JavaScript que contiene el código que debe ejecutar el *Worker*.

```
var worker = new Worker('my_worker.js');
```

De esta manera tenemos disponible y listo para utilizar un nuevo *Worker*. En este momento, podríamos pensar que podemos llamar a métodos o utilizar objetos definidos dentro del nuevo *Worker*, pero no nada más lejos de la realidad. La única manera de comunicarnos con el nuevo *Worker* es a través del paso de mensajes, como hemos visto anteriormente.

```
worker.postMessage('Hello World');
```

Éste método únicamente acepta un parámetro, la cadena de texto a enviar al *Worker*. Por otra parte, la manera de recibir mensajes originados en el *Worker* es definiendo un escuchador para el evento `message`. Los datos incluidos por el *Worker* se encuentran disponibles en la propiedad `data` del evento.

```
worker.addEventListener('message', function(e) {
    alert(e.data);
}, false);
```

12.2.1 DENTRO DE UN *WORKER*

Evidentemente, dentro de un *Worker* necesitamos comunicarnos con el *thread* principal, tanto para recibir los datos de los mensajes como para nuevos datos de vuelta. Para ello, añadimos un escuchador para el evento `message`, y enviamos los datos de vuelta utilizando el mismo método `postMessage`.

```
this.addEventListener('message', function(e) {
    postMessage("I'm done!");
});
```

Es conveniente saber, que a diferencia de la ejecución un *script* en el documento principal, la visibilidad de un *Worker* es mucho más reducida. En concreto, la palabra

reservada `this` no hace referencia al objeto `window`, sino al *Worker* en sí mismo. Debido al comportamiento de ejecución en paralelo de los *Web workers*, éstos solo pueden acceder al siguiente conjunto de funciones de JavaScript (según la especificación):

- Enviar datos con `postMessage` y aceptar mensajes entrantes a través del evento `onmessage`.
- `close`, para terminar con el *Worker* actual.
- Realizar peticiones Ajax.
- Utilizar las funciones de tiempo `setTimeout()/clearTimeout()` y `setInterval()/clearInterval()`.
- Las siguientes funciones de JavaScript: `eval`, `isNaN`, `escape`, etc.
- `WebSockets`.
- `EventSource`.
- Bases de datos `Web SQL`, `IndexedDB`.
- `Web Workers`.

En cambio, los *Workers* **NO** pueden acceder a las siguientes funciones:

- `DOM` (no es seguro para el subproceso).
- Objeto `window`.
- Objeto `document`.
- Objeto `parent`.

12.3 SUBWORKERS

Los *Workers* tienen la capacidad de generar *Workers* secundarios. Esto significa, que podemos dividir la tarea principal en subtareas, y crear nuevos *Workers* dentro del *Worker* principal. Sin embargo, a la hora de utilizar los *Subworkers*, y antes de poder devolver el resultado final al hilo principal, es necesario asegurarse que todos los procesos han terminado.

```
var pendingWorkers = 0, results = {},;
```

```
onmessage = function (event) {
    var data = JSON.parse(event.data), worker = null;
    pendingWorkers = data.length;
```

```
    for (var i = 0; i < data.length; i++) {
        worker = new Worker('subworker.js');
        worker.postMessage(JSON.stringify(data[i]));
        worker.onmessage = storeResult;
```

```
    }
}
```

```
function storeResult(event) {
    var result = JSON.parse(event.data);
```

```
    pendingWorkers--;
    if (pendingWorkers <= 0) {
        postMessage(JSON.stringify(results));
    }
}
```

12.4 GESTIONAR ERRORES

Si se produce un error mientras se ejecuta un Worker, se activa un evento `error`. La interfaz incluye tres propiedades útiles para descubrir la causa del error: `filename` (el nombre de la secuencia de comandos del Worker que causó el error), `lineno` (el número de línea donde se produjo el error) y `message` (una descripción significativa del error).

Ejemplo: `workerWithError.js` intenta ejecutar `1/x`, donde el valor de `x` no se ha definido:

```
<output id="error" style="color: red;"></output>
<output id="result"></output>
```

```
<script>
```

```
function onError(e) {
    document.getElementById('error').textContent = [
        'ERROR: Line ', e.lineno, ' in ', e.filename, ': ', e.message].join('')
;
}
```

```
function onMsg(e) {
    document.getElementById('result').textContent = e.data;
}
```

```
var worker = new Worker('workerWithError.js');
worker.addEventListener('message', onMsg, false);
worker.addEventListener('error', onError, false);
worker.postMessage(); // Start worker without a message.
```

```
</script>
```

workerWithError.js:

```
self.addEventListener('message', function(e) {
    postMessage(1/x); // Intentional error.
};
```

12.5 SEGURIDAD

Debido a las restricciones de seguridad de Google Chrome (otros navegadores no aplican esta restricción), los Workers no se ejecutarán de forma local (por ejemplo, desde `file://`) en las últimas versiones del navegador. En su lugar, fallan de forma automática. Para ejecutar tu aplicación desde el esquema `file://`, ejecuta Chrome con el conjunto de marcadores `--allow-file-access-from-files`.

Las secuencias de comandos del Worker deben ser archivos externos con el mismo esquema que su página de llamada. Por ello, no se puede cargar una secuencia de comandos desde una URL `data:` o una URL `javascript:`. Asimismo, una página `https:` no puede iniciar secuencias de comandos de Worker que comiencen con una URL `http:`.

Ejercicio 14

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 13 WEBSOCKETS

13.1 INTRODUCCIÓN

Internet se ha creado en gran parte a partir del llamado paradigma solicitud/respuesta de HTTP. Un cliente carga una página web, se cierra la conexión y no ocurre nada hasta que el usuario hace clic en un enlace o envía un formulario.

Hace ya algún tiempo que existen tecnologías que permiten al servidor enviar datos al cliente en el mismo momento que detecta que hay nuevos datos disponibles. Se conocen como "Push" o "Comet". Uno de los trucos más comunes para crear la ilusión de una conexión iniciada por el servidor se denomina *Long Polling*. Con el *Long Polling*, el cliente abre una conexión HTTP con el servidor, el cual la mantiene abierta hasta que se envíe una respuesta. Cada vez que el servidor tenga datos nuevos, enviará la respuesta. El *Long Polling* y otras técnicas funcionan bastante bien y de hecho ha sido utilizadas en muchas aplicaciones como el chat de Gmail.

Los *WebSockets* nos ofrecen una conexión bidireccional entre el servidor y el navegador. Esta conexión se produce en tiempo real y se mantiene permanentemente abierta hasta que se cierre de manera explícita. Esto significa que cuando el servidor quiere enviar datos al servidor, el mensaje se traslada inmediatamente. Efectivamente, esto es lo que sucedía al utilizar *tecnologías* como Comet, pero se conseguía utilizando una serie de *trucos*. Si esto no funcionada, siempre era posible utilizar Ajax para conseguir un resultado parecido, pero sobrecargando el servidor de manera innecesaria.

Si disponemos de un socket abierto, el servidor puede enviar datos a todos los clientes conectados a ese socket, sin tener que estar constantemente procesando peticiones de Ajax. La ventaja en cuanto a rendimiento y escalabilidad es bastante evidente al utilizar *WebSockets*.

La latencia en las comunicaciones es otro de los beneficios de utilizar *WebSockets*. Como el socket está siempre abierto y escuchando, los datos son enviados inmediatamente desde el servidor al navegador, reduciendo el tiempo al mínimo, en comparación con un paradigma basado en Ajax, donde hay que realizar una petición, procesar la respuesta y enviarla de nuevo de vuelta.

Finalmente, los datos a transmitir se reducen también de manera drástica, pasando de un mínimo de 200-300 bytes en peticiones Ajax, a 10-20 bytes utilizando websockets.

13.2 CREAR UN WEBSOCKET

El API de WebSocket es realmente sencillo de utilizar. Actualmente, los navegadores únicamente soportan el envío de cadenas de caracteres, y se realiza de una manera muy similar a la que utilizábamos para enviar mensajes en los *Web Workers*. El API está limitado a métodos para abrir la conexión, enviar y recibir datos y cerrar la conexión.

Para abrir una conexión `WebSocket`, sólo tenemos que ejecutar el constructor WebSocket, que toma como parámetro la URL del socket a abrir. Hay que tener en cuenta que el protocolo a utilizar es `ws://`:

```
var socket = new WebSocket('ws://html5rocks.websocket.org/tweets');
```

También existe un protocolo `wss://` para conexiones *WebSocket* seguras, de la misma forma que se utiliza `https://` para las conexiones `HTTP` seguras.

La URL que utilizamos para conectarnos con el *WebSocket* no tiene por qué pertenecer al mismo dominio que nuestro documento, por lo que podemos conectarnos a servicios de terceros sin problemas, expandiendo las posibilidades de nuestra aplicación.

13.3 COMUNICACIÓN CON EL SERVIDOR

Cuando se establece una conexión con el servidor (cuando el evento `open` se activa), se puede empezar a enviar datos al servidor con el método `send` a través del socket creado.

```
// Send new Tweet
socket.send("Hey there, I'm using WebSockets");
```

De la misma forma, el servidor puede enviarnos mensajes en cualquier momento. Cada vez que esto ocurra, se activa el evento `onmessage`. Los datos enviados por el servidor se encuentran en la propiedad `data` del objeto `event`.

```
socket.onmessage = function(event) {
    var data = JSON.parse(event.data);
    if (data.action == 'joined') {
        initiliseChat();
    } else {
        showNewMessage(data.who, data.text);
    }
});
```

El API incorpora además dos eventos que se disparan cuando el socket se abre y está listo, y cuando éste se va a cerrar:

```
socket.onopen  = function(e){ log("Welcome - status "+this.readyState); };
socket.onclose = function(e){ log("Disconnected - status "+this.readyState); };
```

13.4 WEBSOCKET EN EL SERVIDOR

Al utilizar los *WebSocket*, se crea un patrón de uso completamente nuevo para las aplicaciones de servidor. Aunque las pilas de servidor tradicionales como `LAMP` están diseñadas a partir del ciclo de solicitud-respuesta de `HTTP`, a menudo dan problemas si hay muchas conexiones `WebSocket` abiertas. Mantener un gran número de conexiones abiertas de forma simultánea requiere una arquitectura capaz de recibir un alto nivel de concurrencia sin consumir muchos recursos. Estas arquitecturas suelen estar basadas en subprocesos o sistemas de `E/S` asíncronos.

En el próximo capítulo sobre *Server-Sent Events*, veremos una implementación de un servidor web basado en JavaScript, llamado Node.js.

Ejercicio 15

Ver enunciado

ÍNDICE DE CONTENIDOS

CAPÍTULO 14 EVENTSOURCE

Los *EventSource* (también conocidos como *Server-Sent Events*), son eventos en tiempo real transmitidos por el servidor y recibidos en el navegador. Son similares a los *WebSockets* en que suceden el tiempo real, pero son principalmente un método de comunicación unidireccional desde el servidor. Al igual que en los *WebSocket*, creamos una nueva conexión indicando la URL, y el navegador intentará conectarse inmediatamente. El objeto `EventSource` dispone de los siguientes eventos:

- `open`: se dispara cuando la conexión se ha establecido.
- `message`: evento que indica la llegada de un mensaje nuevo.
- `error`: se dispara cuando algo ha ido mal.

Lo que hace a `EventSource` diferente es la manera en que controla las pérdidas de conexión y la gestión de los mensajes.

Si la conexión se pierde por alguna razón, el API automáticamente trata de volver a conectarse. Además, al restablecer la conexión, el cliente envía al servidor la `ID` del último mensaje que recibió. Esto permite al servidor, enviar al cliente todos los mensajes que no ha podido recibir. No es necesario realizar ninguna configuración especial en nuestro código, simplemente el servidor nos enviará los mensajes que no hemos recibido.

Un sencillo ejemplo:

```
var es = new EventSource('/bidding');

es.onopen = function () {
    initialiseData();
};

es.onmessage = function (event) {
    var data = JSON.parse(event.data);
    updateData(data.time, data.bid);
};
```

14.1 EVENTSOURCE EN EL SERVIDOR

En el lado del servidor podemos seguir utilizando una solución basada en PHP y la pila completa LAMP, pero como Apache no se comporta de manera estable con conexiones

persistentes, constantemente trata de cerrar las conexiones y `EventSource` trata de volver a conectarse automáticamente. Esto da como resultado un comportamiento más parecido a Ajax que a una comunicación unidireccional y en tiempo real desde el servidor.

Realmente, esta no es la mejor manera de aprovechar las ventajas de `EventSource`. Para ello, necesitamos una conexión persistente con el servidor, y LAMP no nos lo puede proporcionar. Actualmente existen soluciones de servidor basadas en eventos, como pueden ser Node.js (un servidor basado en JavaScript) o Twisted para Python.

14.1.1 UN SIMPLE SERVIDOR PARA `EVENTSOURCE`

El siguiente código muestra como crear un servidor muy simple con Node.js, el cual acepta conexiones y envía mensajes a los clientes conectados. En este caso, únicamente se notifica al resto de usuarios conectados al servicio, que un nuevo usuario se ha conectado.

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/event-stream',
                        'Cache-Control': 'no-cache'});
    // get the last event id and convert to a number
    var lastId = req.headers['last-event-id']*1;
    if (lastId) {
        for (var i = lastId; i < eventId; i++) {
            res.write('data: ' + JSON.stringify(history[eventId])
                                + '\nid: ' + eventId + '\n');
        }
    }

    // finally cache the response connection
    connections.push(res);

   // When a regular web request is received
    connections.forEach(function (response) {
        history[++eventId] = { agent: req.headers['user-agent'],
                               time: +new Date };
        res.write('data: ' + JSON.stringify(history[eventId])
                           + '\nid: ' + eventId + '\n');
    });
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

En el lado del cliente, el código sería tan sencillo como el siguiente:

```
var es = new EventSource('/eventsource');
es.onmessage = function (event) {
```

```
    var data = JSON.parse(event.data);
    log.innerHTML += '<li><strong>' + data.agent
                    + '</strong><br> connected at <em>'
                    + (new Date(data.time)) + '</em></li>';
};
```

Una *aplicación* muy simple, pero que nos da una idea del funcionamiento de los eventos en tiempo real, utilizando un servidor basado en eventos.

ÍNDICE DE CONTENIDOS

CAPÍTULO 15 FILE

HTML5 ofrece una forma estándar de interactuar con archivos locales a través de la especificación del API de archivos. El API de archivos se puede utilizar, por ejemplo, para crear una vista previa en miniatura de imágenes mientras se envían al servidor o para permitir que una aplicación guarde una referencia de un archivo mientras el usuario se encuentra sin conexión. También se podría utilizar para verificar si el tipo MIME de un archivo seleccionado por el usuario coincide con los formatos de archivo permitidos o para restringir el tamaño de un fichero, antes de enviarlo al servidor.

A continuación se indican las *interfaces* que ofrece la especificación para acceder a archivos desde un sistema de archivos local:

- `File`: representa un archivo local y proporciona información únicamente de lectura (el nombre, el tamaño del archivo, el tipo MIME y una referencia al manejador del archivo).
- `FileList`: representa un conjunto de objetos `File` (tanto para un conjunto de ficheros seleccionados a través de `<input type="file" multiple>` como para un conjunto de ficheros arrastrados desde el sistema de ficheros al navegador).
- `Blob`: permite fragmentar un archivo en intervalos de *bytes*.

Cuando se utiliza junto con las estructuras de datos anteriores, el API de `FileReader` se puede utilizar para leer un archivo de forma asíncrona mediante el control de eventos de JavaScript. Por lo tanto, se puede controlar el progreso de una lectura, detectar si se han producido errores y determinar si ha finalizado una carga de un fichero. El modelo de evento de `FileReader` guarda muchas semejanzas con el API de `XMLHttpRequest`.

15.1 DETECCIÓN DE LA FUNCIONALIDAD

En primer lugar, se debe comprobar que el navegador sea totalmente compatible con el API de archivos. Si la aplicación solo va a utilizar algunas funcionalidades del API, se debe modificar este fragmento de código para adaptarlo a nuestras necesidades:

```
// Check for the various File API support.
if (window.File && window.FileReader && window.FileList && window.Blob) {
    // Great success! All the File APIs are supported.
} else {
    alert('The File APIs are not fully supported in this browser.');
}
```

15.2 ACCESO A TRAVÉS DEL FORMULARIO

La forma más sencilla de cargar un archivo es utilizar un elemento `<input type="file">` estándar de formulario. JavaScript devuelve la lista de objetos `File` seleccionados como un objeto `FileList`.

A continuación, se muestra un ejemplo en el que se utiliza el atributo `multiple` para permitir la selección simultánea de varios archivos:

```
<input type="file" id="files" name="files[]" multiple />
<output id="list"></output>
```

```
function handleFileSelect(e) {
    var files = e.target.files; // FileList object
```

```
    // files is a FileList of File objects. List some properties.
    var output = [];
    for (var i = 0, f; f = files[i]; i++) {
        output.push('<li><strong>', escape(f.name), '</strong> (', f.type || 'n
/a', ') - ',
                    f.size, ' bytes, last modified: ',
                    f.lastModifiedDate.toLocaleDateString(), '</li>');
    }
    list.innerHTML = '<ul>' + output.join('') + '</ul>';
}

files.addEventListener('change', handleFileSelect, false);
```

15.3 CÓMO LEER ARCHIVOS

Después de obtener una referencia de `File`, podemos crear una instancia de un objeto `FileReader` para leer su contenido y almacenarlo en memoria. Cuando finaliza la carga, se lanza el evento `onload` y se puede utilizar su atributo `result` para acceder a los datos del archivo.

A continuación se indican las cuatro opciones de lectura asíncrona de archivo que incluye `FileReader`:

- `FileReader.readAsBinaryString(Blob|File)`: la propiedad `result` contendrá los datos del archivo/objeto `BLOB` en forma de cadena binaria. Cada *byte* se representa con un número entero comprendido entre `0` y `255`, ambos incluidos.
- `FileReader.readAsText(Blob|File, opt_encoding)`: la propiedad `result` contendrá los datos del archivo/objeto `BLOB` en forma de cadena de texto. De forma predeterminada, la cadena se decodifica con el formato `UTF-8`. Podemos especificar un parámetro de codificación opcional para especificar un formato diferente.

- `FileReader.readAsDataURL(Blob|File)`: la propiedad `result` contendrá los datos del archivo/objeto `BLOB` codificados como una URL de datos.
- `FileReader.readAsArrayBuffer(Blob|File)`: la propiedad `result` contendrá los datos del archivo/objeto `BLOB` como un objeto `ArrayBuffer`.

Una vez que se ha activado uno de estos métodos de lectura en el objeto `FileReader`, se pueden escuchar los eventos `onloadstart`, `onprogress`, `onload`, `onabort`, `onerror` y `onloadend` para realizar un seguimiento de su progreso de lectura. En el ejemplo que se muestra a continuación, obtenemos las imágenes de los elementos seleccionados por el usuario, leemos su contenido con `reader.readAsDataURL()` mostramos una miniatura de la imagen:

```
<input type="file" id="files" name="files[]" multiple />
<output id="list"></output>
```

```
function handleFileSelect(evt) {
    var files = evt.target.files; // FileList object
```

```
    // Loop through the FileList and render image files as thumbnails.
    for (var i = 0, f; f = files[i]; i++) {

        // Only process image files.
        if (!f.type.match('image.*')) {
            continue;
        }

        var reader = new FileReader();
        // Closure to capture the file information.
        reader.onload = (function(theFile) {
            return function(e) {
                // Render thumbnail.
                var span = document.createElement('span');
                span.innerHTML = ['<img class="thumb" src="', e.target.result,
                                  '" title="', escape(theFile.name), '"/>'].joi
n('');
                list.insertBefore(span, null);
            };
        })(f);

        // Read in the image file as a data URL.
        reader.readAsDataURL(f);
    }
}
```

```
files.addEventListener('change', handleFileSelect, false);
```

15.4 FRAGMENTACIÓN DE ARCHIVOS

En algunos casos, leer el archivo completo en memoria no es la mejor opción. Supongamos, por ejemplo, que se quiere crear una herramienta de subida de archivos de forma asíncrona. Para acelerar la subida, se podría leer y enviar el archivo en diferentes fragmentos de *bytes*. El servidor se encargaría de reconstruir el contenido del archivo en el orden correcto.

Afortunadamente, la interfaz `File` nos proporciona un método de fragmentación de ficheros. El método utiliza un *byte* de inicio como primer argumento, un *byte* de finalización como segundo argumento. El siguiente ejemplo muestra cómo leer un fichero por partes:

```
var files = document.getElementById('files').files;
if (!files.length) {
  alert('Please select a file!');
  return;
}

var file  = files[0];
var start = 0;
var stop  = file.size - 1;

var reader = new FileReader();

reader.onloadend = function(e) {
  if (e.target.readyState == FileReader.DONE) { // DONE == 2
    document.getElementById('byte_content').textContent = e.target.result;
    document.getElementById('byte_range').textContent =
        ['Read bytes: ', start + 1, ' - ', stop + 1,
         ' of ', file.size, ' byte file'].join('');
  }
};

if (file.webkitSlice) {
  var blob = file.webkitSlice(start, stop + 1);
} else if (file.mozSlice) {
  var blob = file.mozSlice(start, stop + 1);
}
reader.readAsBinaryString(blob);
```

ÍNDICE DE CONTENIDOS

CAPÍTULO 16 HISTORY

El API `history` de HTML es la manera estándar de manipular el historial de navegación a través de JavaScript. Partes de esta API ya se encontraban disponibles en versiones anteriores de HTML. Ahora, HTML5 incluye una nueva manera de añadir entradas al historial de navegación, modificando la URL pero sin actualizar la página actual, y eventos que se disparan cuando el usuario a eliminado estas entradas, pulsando el botón de volver del navegador. Esto quiere decir que la barra de direcciones sigue funcionando de la misma manera, identificando los recursos de manera única, aún cuando las aplicaciones hacen un uso intensivo de JavaScript sin recargar la página.

Como sabemos, una URL representa un recurso único. Podemos enlazarlo directamente, almacenarlo como favorito, los motores de búsqueda pueden analizar su contenido, podemos copiarlo y enviarlo por email... La URL realmente importa.

Así pues, lo que queremos es que contenidos únicos dispongan de una URL única. Hasta ahora, el comportamiento normal de los navegadores recargar de nuevo la página si modificábamos la URL, realizando una nueva petición y obteniendo de nuevo todos los recursos del servidor. No había manera de decir al navegador que cambiase la URL pero descargase únicamente la mitad de la página. El API `history` de HTML5 permite precisamente esto. En lugar de solicitar la carga de toda la página, podemos utilizar JavaScript para cargar únicamente los contenidos que deseemos.

La idea es la siguiente. Imaginemos que tenemos una página A y otra página B, que comparten el 90% de su contenido. Cuando un usuario se encuentra en la página A, y quiere navegar a la B, lo normal es realizar una petición completa. En lugar de realizar esta petición, interrumpimos esta navegación y realizamos los siguientes pasos de manera manual:

1. Cargar el 10% de contenido diferente de la página B, a través de algún método como AJAX.
2. Cambiar el contenido, utilizando `innerHTML` u otros métodos del DOM. Es posible que tengamos que reiniciar los eventos asociados a los elementos.
3. Actualizamos la URL del navegador, indicando la dirección de la página B, utilizando el API `history` de HTML5.

Tras realizar estos pasos, disponemos de un DOM exacto al de la página B, como si hubiésemos navegado hasta ella, pero sin realizar una petición completa.

16.1 API

El API de HTML4 ya incluía algunos métodos básicos para movernos a través del historial de navegación, como eran `history.back();`, `history.forward()` y `history.go(n)`. Sin embargo, estos métodos no permitían modificar la pila del historial, por lo que no eran de gran utilidad. HTML5 ha introducido dos nuevos métodos que nos permiten añadir y modificar las entradas del historial, concretamente `history.pushState()` y `history.replaceState()`. Además de estos métodos, se ha añadido también un evento `window.onpopstate`, que es lanzado cada vez que alguna de las entradas de `history` cambia.

16.1.1 MÉTODO `PUSHSTATE()`

Supongamos que estamos visitando `http://www.arkaitzgarro.com/html5/index.html` y a través de un *script* realizamos la siguiente operación:

```
var stateObj = { foo: "bar" };
history.pushState(stateObj, "Demos", "demos.html");
```

Esto va a provocar que en la barra de direcciones se muestre `http://www.arkaitzgarro.com/html5/demos.html`, pero el navegador no va a cargar `demos.html` ni va a comprobar su existencia. En este punto, si navegamos a otra página como `http://www.google.es/`, y después presionamos el botón de volver, en la URL se mostrará `http://www.arkaitzgarro.com/html5/demos.html` y la página lanzará el evento `popstate`, donde el *estado* contiene una copia de `stateObj`. La página que se mostrará será `index.html`, pero deberemos realizar un trabajo extra al lanzarse el evento para mostrar el contenido correcto.

El método `pushState()` toma tres parámetros: un objeto de *estado*, un título y una URL:

- El objeto de **estado**: es un objeto de JavaScript asociado con la nueva entrada del historial creada con `pushState()`. Cada vez que el usuario navega al estado creado, el evento `popstate` es disparado, y la propiedad `state` del evento contiene una copia de este objeto. Este objeto puede representar cualquier que se pueda serializar. Como este objeto se almacena en el disco, es posible recuperarlo aunque el navegador se cierre. Los navegadores imponen un límite de tamaño a la hora de almacenar estados (en el caso de Firefox `640KB`). Si se necesita más espacio, es recomendable utilizar `sessionStorage` o `localStorage`.
- El **título**: representa el nuevo título de la página a la que *navegamos*.
- La nueva **URL**: corresponde con la nueva URL que se añade al historial de navegación. Esta URL puede ser absoluta o relativa, la única restricción es que corresponda al dominio del documento actual. Si no se especifica este parámetro, la URL corresponde con el documento actual.

En esencia, ejecutar el método `pushState()` es similar a definir `window.location = "#foo"`, ya que en ambos casos se crea y activa una nueva entrada en el historial asociada con el documento actual. Pero `pushState()` ofrece las siguientes ventajas:

- La nueva URL puede ser cualquier URL dentro del dominio actual. En cambio, `window.location = "#foo"` se mantiene siempre en el documento actual.
- No es necesario cambiar la URL actual para añadir una nueva entrada y almacenar datos asociados.
- Podemos asociar datos a una nueva entrada en el historial. Con el enfoque basado en `hash` (#), los datos tenemos que añadirlos a la URL.

16.1.2 MÉTODO `REPLACESTATE()`

El método `history.replaceState()` funciona de manera similar a `history.pushState()`, a excepción de que `replaceState()` modifica la entrada actual del historial, en lugar de añadir una nueva. Éste método es útil cuando queremos actualizar el objeto de estado de la entrada actual en respuesta a una acción del usuario.

16.1.3 EVENTO `POPSTATE`

El evento `popstate` es lanzado cada vez que la entrada actual del historial cambia por otra entrada existente en el mismo documento. Si la entrada del historial que está siendo activada fue creada a través de `history.pushState()` o se modificó con `history.replaceState()`, el evento `popstate` recibe como parámetro una copia del estado de la entrada del historial.

Éste evento no se lanza cuando se llama a `history.pushState()` o `history.replaceState()`. Únicamente se dispara realizando una acción en el navegador como pulsando el botón *atrás* o ejecutando `history.back()` en JavaScript. Un ejemplo de este comportamiento:

```
window.onpopstate = function(event) {
    alert(document.location + ", state: " + JSON.stringify(event.state));
};

history.pushState({page: 1}, "title 1", "?page=1");
history.pushState({page: 2}, "title 2", "?page=2");
history.replaceState({page: 3}, "title 3", "?page=3");

// alerts "http://example.com/index.html?page=1, state: {"page":1}"
history.back();
// alerts "http://example.com/index.html, state: null
history.back();
// alerts "http://example.com/index.html?page=3, state: {"page":3}
history.go(2);
```

Cuando la página se carga, por ejemplo al reiniciar el navegador, tango Chrome como Safari lanzan el evento `popstate`, pero no es el caso de Firefox. Sin embargo, en este caso, es posible acceder a los datos almacenados en `pushState` desde la propiedad `state` del objeto `history`.

```
window.onload = function() {
    var currentState = history.state;
}
```

ÍNDICE DE CONTENIDOS

CAPÍTULO 17 XMLHTTPREQUEST2

XMLHttpRequest forma parte de las mejoras incrementales que los creadores de navegadores están implementando a la plataforma base.

XMLHttpRequest de nivel 2 introduce una gran cantidad de nuevas funciones que ponen fin a los problemas de nuestras aplicaciones web, como solicitudes de origen cruzado, eventos de progreso de subidas y compatibilidad con subida/bajada de datos binarios. Esto permite a AJAX trabajar en coordinación con muchas de las API HTML5 más punteras, como API de FileSystem, el API de Web Audio y WebGL.

17.1 RECUPERACIÓN DE ARCHIVOS

Recuperar archivos como `blob` binario era muy complicado con `XHR`. Técnicamente, no era ni siquiera posible. Un truco que se ha documentado mucho implicaba anular el tipo mime con un conjunto de caracteres definido por el usuario, como se muestra a continuación.

La antigua forma de recuperar una imagen:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
```

```
// Hack to pass bytes through unprocessed.
xhr.overrideMimeType('text/plain; charset=x-user-defined');
```

```
xhr.onreadystatechange = function(e) {
    if (this.readyState == 4 && this.status == 200) {
        var binStr = this.responseText;
            for (var i = 0, len = binStr.length; i < len; ++i) {
                var c = binStr.charCodeAt(i);
                //String.fromCharCode(c & 0xff);
                var byte = c & 0xff;  // byte at offset i
            }
        }
    };
```

```
xhr.send();
```

Aunque funciona, lo que se obtiene realmente en `responseText` no es un `blob` binario,

sino una cadena binaria que representa el archivo de imagen. Estamos engañando al servidor para que devuelva los datos sin procesar.

17.2 ESPECIFICACIÓN DE UN FORMATO DE RESPUESTA

En lugar de realizar la acción del ejemplo anterior, vamos a aprovechar las nuevas propiedades de XMLHttpRequest (`responseType` y `response`) para indicar al navegador el formato en el que queremos que nos devuelva los datos.

- `xhr.responseType`: antes de enviar una solicitud, establece `xhr.responseType` en `text`, `arraybuffer`, `blob` o `document`, en función de los datos que necesites. Ten en cuenta que si se establece `xhr.responseType = ''` (o si se omite), se utilizará la respuesta predeterminada `text`.
- `xhr.response`: después de una solicitud correcta, la propiedad `response` de `xhr` contendrá los datos solicitados como `DOMString`, `ArrayBuffer`, `Blob` o `Document` (en función del valor establecido para responseType).

Con esto, podemos recuperar la imagen como `ArrayBuffer` en lugar de como una cadena. Al transferir el búfer al API `BlobBuilder` se crea un `Blob`:

```
BlobBuilder = window.MozBlobBuilder || window.WebKitBlobBuilder || window.BlobB
uilder;
```

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'arraybuffer';
```

```
xhr.onload = function(e) {
    if (this.status == 200) {
        var bb = new BlobBuilder();
        bb.append(this.response); // Note: not xhr.responseText
```

```
        var blob = bb.getBlob('image/png');
        ...
    }
};
```

```
xhr.send();
```

Si se quiere trabajar directamente con un `Blob` o no se necesita manipular ni un solo *byte* del archivo, utilizaremos `xhr`:

```
responseType='blob':
window.URL = window.URL || window.webkitURL;  // Take care of vendor prefixes.
```

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'blob';
```

```
xhr.onload = function(e) {
    if (this.status == 200) {
        var blob = this.response;
```

```
        var img = document.createElement('img');
        img.onload = function(e) {
          window.URL.revokeObjectURL(img.src); // Clean up after yourself.
        };
        img.src = window.URL.createObjectURL(blob);
        document.body.appendChild(img);
        ...
    }
};
```

```
xhr.send();
```

17.3 ENVÍO DE DATOS

Durante algún tiempo, XMLHttpRequest nos ha limitado a enviar datos `DOMString` o `Document` (XML). Pero eso se acabó. Se ha anulado un método `send()` y rediseñado para aceptar todos estos tipos: `DOMString, Document, FormData, Blob, File y ArrayBuffer`.

- Envío de la cadena de datos: `xhr.send(domstring)`

```
sendText('test string'); function sendTextNew(txt) {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', '/server', true);
    xhr.responseType = 'text';
    xhr.onload = function(e) {
        if (this.status == 200) {
            console.log(this.response);
        }
    };
    xhr.send(txt);
}
```

```
sendText2('test string');
```

- Envío de formularios: `xhr.send(formdata)`

```
function sendForm() {
    var formData = new FormData();
    formData.append('username', 'johndoe');
    formData.append('id', 123456);

    var xhr = new XMLHttpRequest();
    xhr.open('POST', '/server', true);
    xhr.onload = function(e) { ... };

    xhr.send(formData);
}
```

Por supuesto, no es necesario crear un objeto `<form>` desde cero. Los objetos `FormData` se pueden inicializar a partir de un elemento `HTMLFormElement` de la página. Por ejemplo:

```
<form id="myform" name="myform" action="/server">
    <input type="text" name="username" value="johndoe">
    <input type="number" name="id" value="123456">
    <input type="submit" onclick="return sendForm(this.form);">
</form>
function sendForm(form) {
    var formData = new FormData(form);

    formData.append('secret_token', '1234567890');

    var xhr = new XMLHttpRequest();
    xhr.open('POST', form.action, true);
    xhr.onload = function(e) { ... };

    xhr.send(formData);

    return false; // Prevent page from submitting.
}
```

Un formulario HTML puede incluir subidas de archivos (como `<input type="file">`) y `FormData` también. Simplemente se añade el archivo o los archivos y el navegador construirá una solicitud multipart/form-data cuando se ejecute `send()`:

```
function uploadFiles(url, files) {
    var formData = new FormData();

    for (var i = 0, file; file = files[i]; ++i) {
    formData.append(file.name, file);
```

```
    }
```

```
    var xhr = new XMLHttpRequest();
    xhr.open('POST', url, true);
    xhr.onload = function(e) { ... };
```

```
    xhr.send(formData);  // multipart/form-data
}
```

```
document.querySelector('input[type="file"]').addEventListener('change', functio
n(e) {
    uploadFiles('/server', this.files);
}, false);
```

17.4 SUBIDA DE ARCHIVOS O BLOB

También podemos enviar datos de `File` o `Blob` con `XHR`.

En este ejemplo se crea un texto nuevo desde cero con el API `lobBuilder y se sube ese`Blob` al servidor. El código también configura un controlador para informar al usuario sobre el progreso de la subida:

```
function upload(blobOrFile) {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', '/server', true);
    xhr.onload = function(e) { ... };
```

```
    // Listen to the upload progress.
    var progressBar = document.querySelector('progress');
    xhr.upload.onprogress = function(e) {
        if (e.lengthComputable) {
            progressBar.value = (e.loaded / e.total) * 100;
            progressBar.textContent = progressBar.value; // Fallback for unsupp
orted browsers.
        }
    };
```

```
    xhr.send(blobOrFile);
}
```

// Take care of vendor prefixes. BlobBuilder = window.MozBlobBuilder || window.WebKitBlobBuilder || window.BlobBuilder;

```
    var bb = new BlobBuilder();
```

```
    bb.append('hello world');
```

```
    upload(bb.getBlob('text/plain'));
```

17.5 SUBIDA DE UN FRAGMENTO DE BYTES

Y por último, podemos enviar `ArrayBuffer` como la carga de XHR.

```
function sendArrayBuffer() {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', '/server', true);
    xhr.onload = function(e) { ... };
```

```
    var uInt8Array = new Uint8Array([1, 2, 3]);
```

```
    xhr.send(uInt8Array.buffer);
}
```

17.6 EJEMPLOS PRÁCTICOS

17.6.1 CÓMO DESCARGAR Y GUARDAR ARCHIVOS EN EL SISTEMA DE ARCHIVOS HTML5

Supongamos que tienes una galería de imágenes y quieres recuperar un grupo de imágenes para, a continuación, guardarlas localmente con el sistema de archivos HTML5. Una forma de conseguir esto sería solicitar imágenes como `ArrayBuffer`, crear un `Blob` a partir de los datos y escribir el blob con `FileWriter`:

```
window.requestFileSystem  = window.requestFileSystem || window.webkitRequestFil
eSystem;
```

```
function onError(e) {
    console.log('Error', e);
}
```

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'arraybuffer';
```

```
xhr.onload = function(e) {
```

```
    window.requestFileSystem(TEMPORARY, 1024 * 1024, function(fs) {
        fs.root.getFile('image.png', {create: true}, function(fileEntry) {
            fileEntry.createWriter(function(writer) {
```

```
            writer.onwrite = function(e) { ... };
            writer.onerror = function(e) { ... };
```

```
            var bb = new BlobBuilder();
            bb.append(xhr.response);
```

```
            writer.write(bb.getBlob('image/png'));
```

```
            }, onError);
        }, onError);
    }, onError);
};
```

```
xhr.send();
```

17.6.2 CÓMO DIVIDIR UN ARCHIVO Y SUBIR CADA FRAGMENTO

Con las API de archivo, podemos minimizar el trabajo necesario para subir un archivo de gran tamaño. La técnica es dividir el archivo que se va a subir en varios fragmentos, crear un `XHR` para cada parte y unir los fragmentos en el servidor. Es similar a la forma en que Gmail sube archivos adjuntos tan grandes en tan poco tiempo.

```
window.BlobBuilder = window.MozBlobBuilder || window.WebKitBlobBuilder ||
                     window.BlobBuilder;
```

```
function upload(blobOrFile) {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', '/server', true);
    xhr.onload = function(e) { ... };
    xhr.send(blobOrFile);
}
```

```
document.querySelector('input[type="file"]').addEventListener('change', functio
n(e) {
    var blob = this.files[0];
```

```
    const BYTES_PER_CHUNK = 1024 * 1024; // 1MB chunk sizes.
    const SIZE = blob.size;
```

```
    var start = 0;
    var end = BYTES_PER_CHUNK;
```

```
    while(start < SIZE) {
```

```
        if ('mozSlice' in blob) {
            var chunk = blob.mozSlice(start, end);
        } else {
            var chunk = blob.webkitSlice(start, end);
        }
```

```
        upload(chunk);
```

```
        start = end;
        end = start + BYTES_PER_CHUNK;
    }
}, false);
```

```
})();
```

ÍNDICE DE CONTENIDOS

CAPÍTULO 18 EJERCICIOS

18.1 CAPÍTULO 2

18.1.1 EJERCICIO 1

Dada la siguiente página web, estructurada en XHTML, aplicar las nuevas etiquetas semánticas de HTML5 donde sea conveniente, manteniendo el mismo aspecto visual (modificar la hoja de estilos si es necesario) y funcionalidad. Realizad también los cambios necesarios en la cabecera del documento (elemento `<head>`).

Descargar página web.

18.2 CAPÍTULO 3

18.2.1 EJERCICIO 2

Crear un formulario que contenga lo siguiente:

- Los 12 nuevos tipos de elementos `input`.
- El nuevo elemento `datalist`, que contenga algunos nombres de provincia y un campo de texto que se relacione con él.
- Una caja de texto (`<input type="text">`), a la cual aplicar los atributos `autofocus`, `placeholder`, `required` y `autocomplete`.
- Una caja de texto que sólo pueda contener números (`<input type="number">`), cuyos valores tienen que estar comprendidos entre 0 y 10.
- Un campo de selección de ficheros (`<input type="file">`), al que aplicar el atributo `multiple`.
- Un campo de introducción de password (`<input type="password">`), donde el valor introducido debe cumplir lo siguiente: debe tener una longitud mínima de 8 caracteres, comenzar por un número y terminar en una letra mayúscula.
- Un nuevo elemento `progress` que represente el avance de completado de campos del formulario.

Acceder al formulario desde al menos 4 navegadores (2 de escritorio y 2 de dispositivos móviles), y comprobad el comportamiento y funcionamiento en cada elemento del formulario. Anotad dichos resultados en una hoja de cálculo para futuras referencias.

18.3 CAPÍTULO 4

18.3.1 EJERCICIO 3

Identificar las siguientes características de al menos 4 navegadores (2 de escritorio y 2 de dispositivos móviles):

- Cuáles de los 12 nuevos tipos de `input` soporta el navegador.
- Qué *codecs* de reproducción de vídeo soporta cada navegador.
- Qué sistema(s) de almacenamiento local soporta cada navegador.

18.3.2 EJERCICIO 4

Identificar si el navegador soporta el atributo `placeholder`. En caso de no soportar dicha funcionalidad, cargar el *polyfill* correspondiente para añadir dicha funcionalidad al navegador.

18.4 CAPÍTULO 5

18.4.1 EJERCICIO 5

A partir del siguiente HTML, realizar los siguientes pasos:

```
<ul>
    <li class="user" data-name="Arkaitz Garro" data-city="Donostia"
        data-lang="es" data-food="Txuleta">Arkaitz Garro</li>
    <li class="user" data-name="John Doe" data-city="Boston"
        data-lang="en" data-food="Bacon">John Doe</li>
    <li class="user" data-name="Divya Resig" data-city="Tokyo"
        data-lang="jp" data-food="Sushi" data-delete="true">Divya Resig</li>
</ul>
```

- Obtener cada uno de los atributos `data-` de los elementos de la lista, y mostrarlos por consola.
- Modificar el idioma `es` por `es_ES`.
- Eliminar los elementos de la lista cuyo atributo `data-delete` sea `true`.

18.5 CAPÍTULO 6

18.5.1 EJERCICIO 6

Crear un reproductor de vídeo que cumple las siguientes características:

- Reproducir los vídeos independientemente del codec soportado por el navegador.
- Incluir controles de reproducción, pausa, parar, avanzar y retroceder 10 segundos, inicio y fin.

- Control de volumen y paso a pantalla completa.
- Un indicador de progreso de la reproducción.

Añadir una lista de reproducción que permita seleccionar un nuevo vídeo, y éste se reproduzca sin recargar la página.

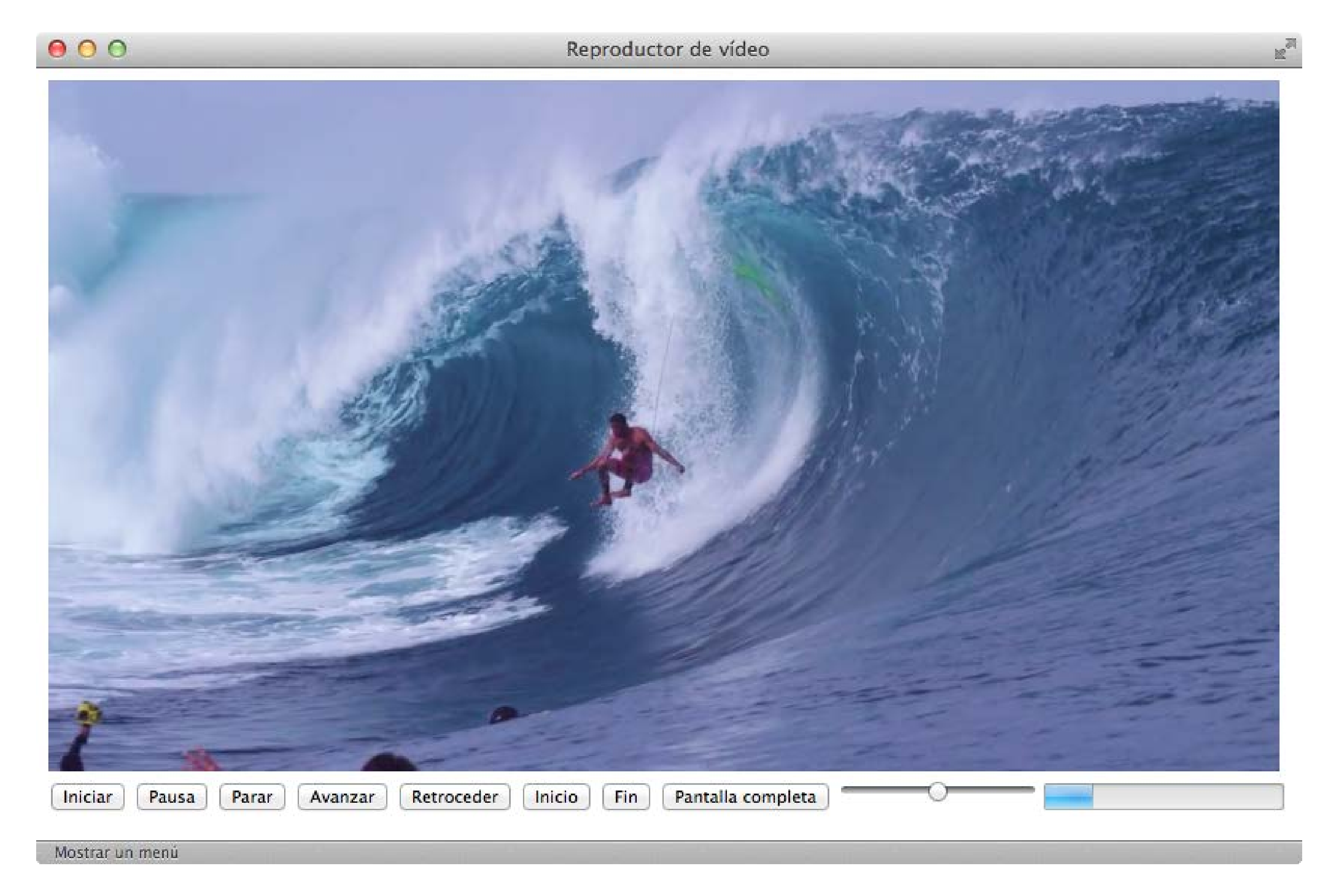


Figura 18.1 Aspecto final del reproductor

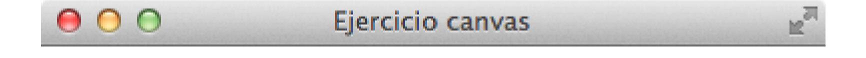
18.6 CAPÍTULO 7

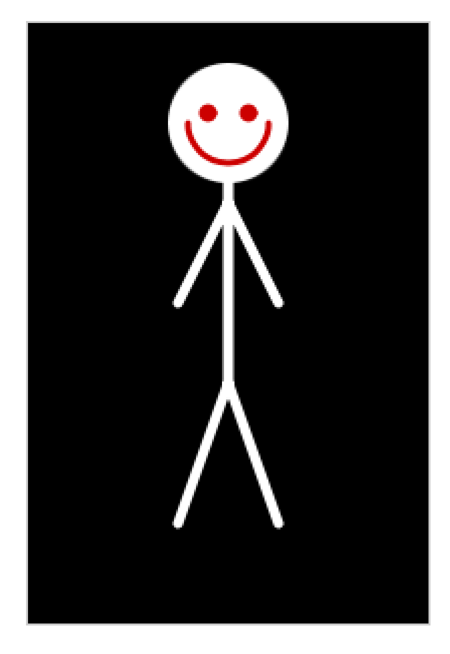
18.6.1 EJERCICIO 7

Dibujar sobre un elemento canvas, un *stickman* con un aspecto final similar al que se muestra a continuación. Seguir las siguiente indicaciones como ayuda:

- El tamaño inicial del canvas es de `200px` del ancho y `300px`de alto.
- Utilizar un rectángulo negro para pintar todo el fondo del canvas.
- Dibujar la cabeza con un arco completo, en las coordenadas (`100,50`), y un radio de `30px`.
- La sonrisa corresponde con un semiarco rojo (`#c00`), con inicio de coordenadas (`100,50`) y un radio de `20px`.
- Los ojos son dos circunferencias rojas situadas en las coordenadas (`90,45`) y

(`110,45`) y de radio `3px`.





Mostrar un menú

Figura 18.2 Resultado final del stickman

18.7 CAPÍTULO 8

18.7.1 EJERCICIO 8

Implementad las siguientes funcionalidades utilizando `SessionStorage` y `LocalStorage`:

- Crear una caja de texto, a modo de editor de contenidos, y utilizando `SessionStorage` almacenar el contenido de la caja en cada pulsación de tecla. Si la página es recargada, el último contenido almacenado debe mostrarse en la caja de texto. Comprobad que cerrando la pestaña actual, o abriendo una nueva ventana, los datos no se comparten.
- Modificar el código anterior para utilizar `LocalStorage`. Comprobad que en este caso, aunque cierre la ventana o abra una nueva, los datos se mantienen. Añadir la posibilidad de actualizar el resto de ventanas abiertas, cada vez que se modifique el valor de la caja de texto en cualquiera de ellas.

18.7.2 EJERCICIO 9

Crear un objeto que encapsule una base de datos WebSQL, que nos permitar acceder a una base de datos para añadir, modificar, eliminar y obtener registros. Dicha base de datos va a almacenar tweets procedentes de Twitter, que tienen asociado el *hashtag* #html5. Los requisitos son los siguientes:

- Disponer de una tabla para almacenar los tweets. Los campos mínimos son: identificador del tweet, texto, usuario, y fecha de publicación.
- Disponer de una tabla para almacenar los usuarios que publican los tweets. Esta tabla debe estar relacionada con la anterior. Los campos mínimos son: identificador del usuario, nombre e imagen.
- Crear un método `addTweet` que dado un objeto que corresponde con un tweet, lo almacene en la base de datos. Almacenar el usuario en caso de que no exista, o relacionarlo con el tweet si existe.
- Crear un método `removeTweet` que dado un identificador de tweet, lo elimine de la base de datos. Éste método debe devolver el tweet eliminado.
- Crear un método `updateTweet` que dado un objeto que corresponde con un tweet, actualice los datos correspondientes al tweet en la base de datos.
- Crear un método `getTweets` que dado un parámetro de fecha, me devuelva todos los tweets posteriores a esa fecha. Cada tweet debe incluir sus datos completos y el usuario que lo creó.

Obtener los últimos 25 tweets que tienen como *hashtag* #html5 de la siguiente consulta a Twitter: http://search.twitter.com/search.json?q=%23html5&rpp=25&result_type="recent"

Consultad la API de Twitter para identificar el formato del resultado, nombres de campos, etc.

18.7.3 EJERCICIO 10

Crear un objeto que encapsule una base de datos IndexedDB, que nos permitar acceder a una base de datos para añadir, modificar, eliminar y obtener registros. Dicha base de datos va a almacenar una sencilla lista de tareas pendientes. Los requisitos son los siguientes:

- Disponer de un *almacén* de tareas pendientes. Sus propiedades son: un identificador único que actúa como índice, el texto descriptivo, una propiedad que nos indique si la tarea está completada o no y la fecha/hora de creación.
- Crear un método `addTask` que dado un objeto que corresponde con una tarea, lo almacene en la base de datos.
- Crear un método `removeTask` que dado un identificador de una tarea, lo elimine de la base de datos. Éste método debe devolver la eliminada.
- Crear un método `updateTask` que dado un identificador de una tarea, actualice los datos correspondientes a la tarea en la base de datos.
- Crear un método `getTasks` que dado un parámetro booelano *completado*, nos devuelva las tareas que se encuentran completadas o no.

18.8 CAPÍTULO 9

18.8.1 EJERCICIO 11

Cread una página HTML que muestre la hora local de vuestro ordenador, y la hora del servidor de www.google.es. Los cálculos de hora local y obtención de hora del servidor, deben crearse en ficheros separados. El comportamiento es el siguiente:

- Todos los ficheros necesarios para realizar el ejercicio deben ser cacheados, a excepción del javascript que solicita la petición al servidor.
- Las peticiones de hora al servidor www.google.es se realizan a través de Ajax, en intervalos de 1 segundo. Dichas peticiones no deben realizarse si no existe conexión. En tal caso, se muestra un mensaje al usuario indicando que se ha perdido la conexión.
- Si se recarga la página, y no existe conexión, utilizar un *fallback* para la obtención de la hora del servidor, que muestre un mensaje al usuario indicando que se ha perdido la conexión.

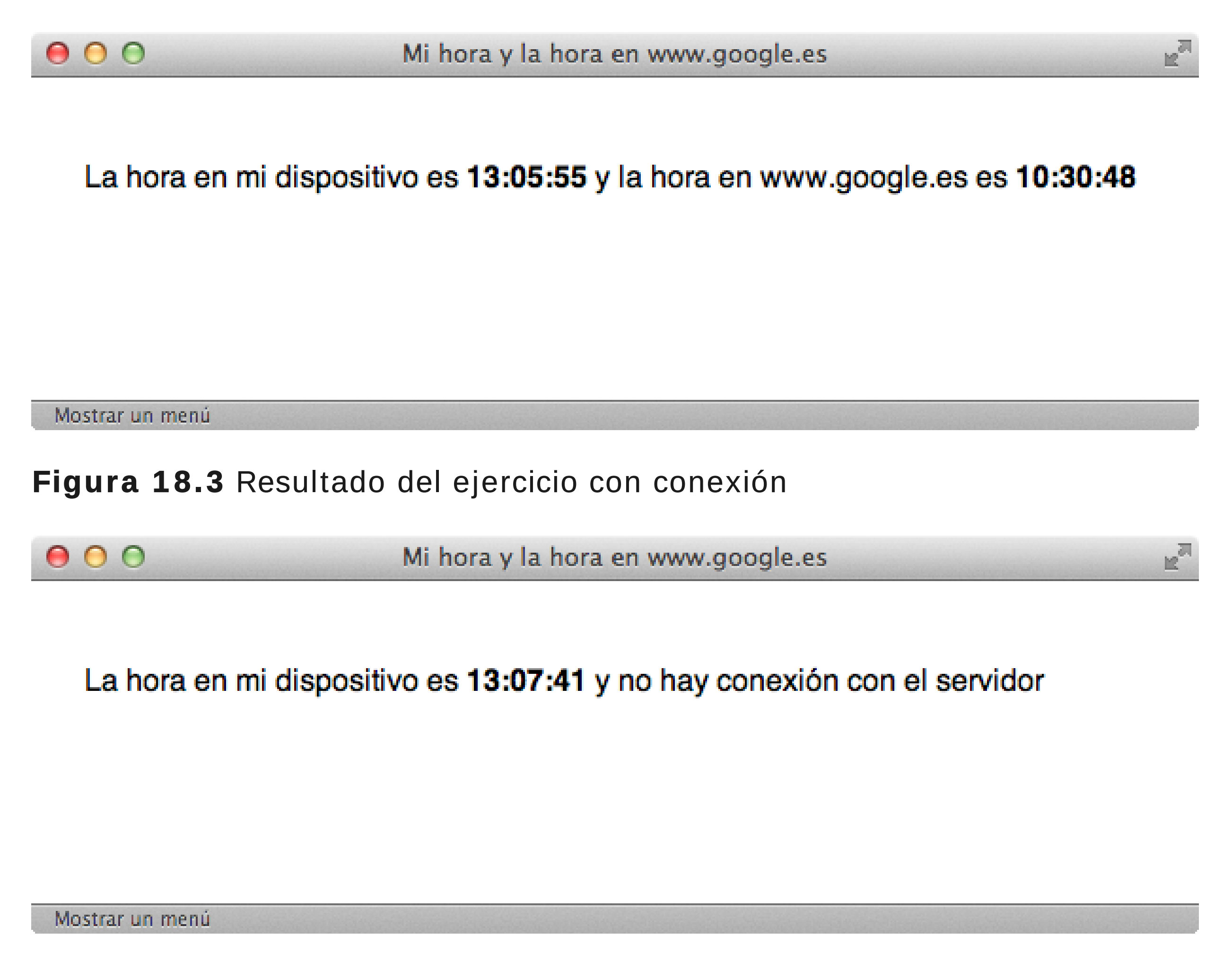


Figura 18.3 Resultado del ejercicio con conexión

Figura 18.4 Resultado del ejercicio sin conexión

Utilizad el siguiente código para obtener la hora de un servidor:

```
function srvTime(url){
```

```
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.open('HEAD',url,false);
    xmlHttp.setRequestHeader("Content-Type", "text/html");
    xmlHttp.send(null);
    return xmlHttp.getResponseHeader("Date");
}

var st = srvTime("https://www.google.es/");
var date = new Date(st);
```

18.9 CAPÍTULO 10

18.9.1 EJERCICIO 12

Implementar un carrito de la compra que permita arrastrar los productos a la cesta, y calcule el precio total de la compra. El proceso es el siguiente:

- Al pasar con el ratón sobre una imagen, el puntero cambia de aspecto (`cursor: move;`) y se permite arrastrar el contenido.
- Al arrastrar el elemento, la zona que contiene la cesta de la compra se ilumina.
- Al soltar la imagen sobre la cesta de la compra, añade el producto si no existía, mostrando la imagen, el nombre del producto, la cantidad de productos en la cesta (en este caso 1) y el coste total para ese producto.
- Al soltar la imagen sobre la cesta de la compra, si el producto ya se encontraba en la cesta, actualiza la cantidad de productos y su precio.

Posibles mejoras para este ejercicio:

- Añadir la posibilidad de eliminar los productos de la cesta de la compra, arrastrándolos desde la propia cesta y soltándolos en cualquier parte de la página. En este caso, mostraríamos un mensaje de confirmación de eliminación del producto.
- Mantener una persistencia de datos para la cesta de la compra, que el actualizar la página no se pierdan los productos introducidos.

Descargar sitio web.

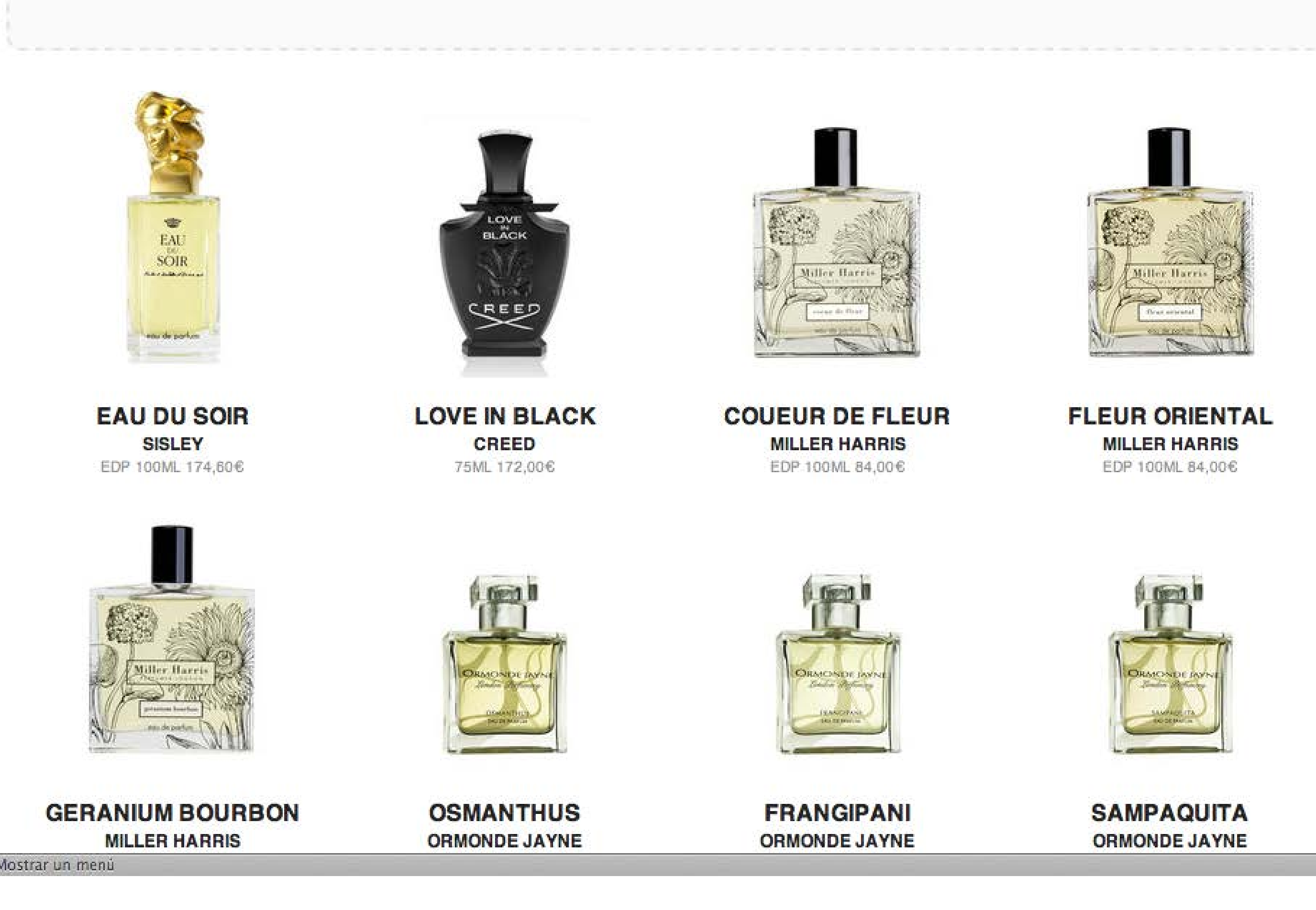


Figura 18.5 Aspecto inicial de la cesta de la compra

18.10 CAPÍTULO 11

18.10.1 EJERCICIO 13

Utilizando los servicios de geolocalización, realizar las siguientes tareas:

- Solicitar las coordenadas actuales, y mostrar dichas coordenadas (y la precisión) tanto en formato texto como un punto en el mapa.
- Haciendo uso del entorno de desarrollo de Android (emulador y SDK) y la herramienta DDMS, indicar al dispositivo una posición GPS concreta. Posteriormente, modificar la implementación para solicitar continuamente la posición del dispositivo. En la herramienta DDMS, utilizar el fichero de rutas GPX que se proporciona y mostrar cada nuevo punto en el mapa.

Descargar sitio web.



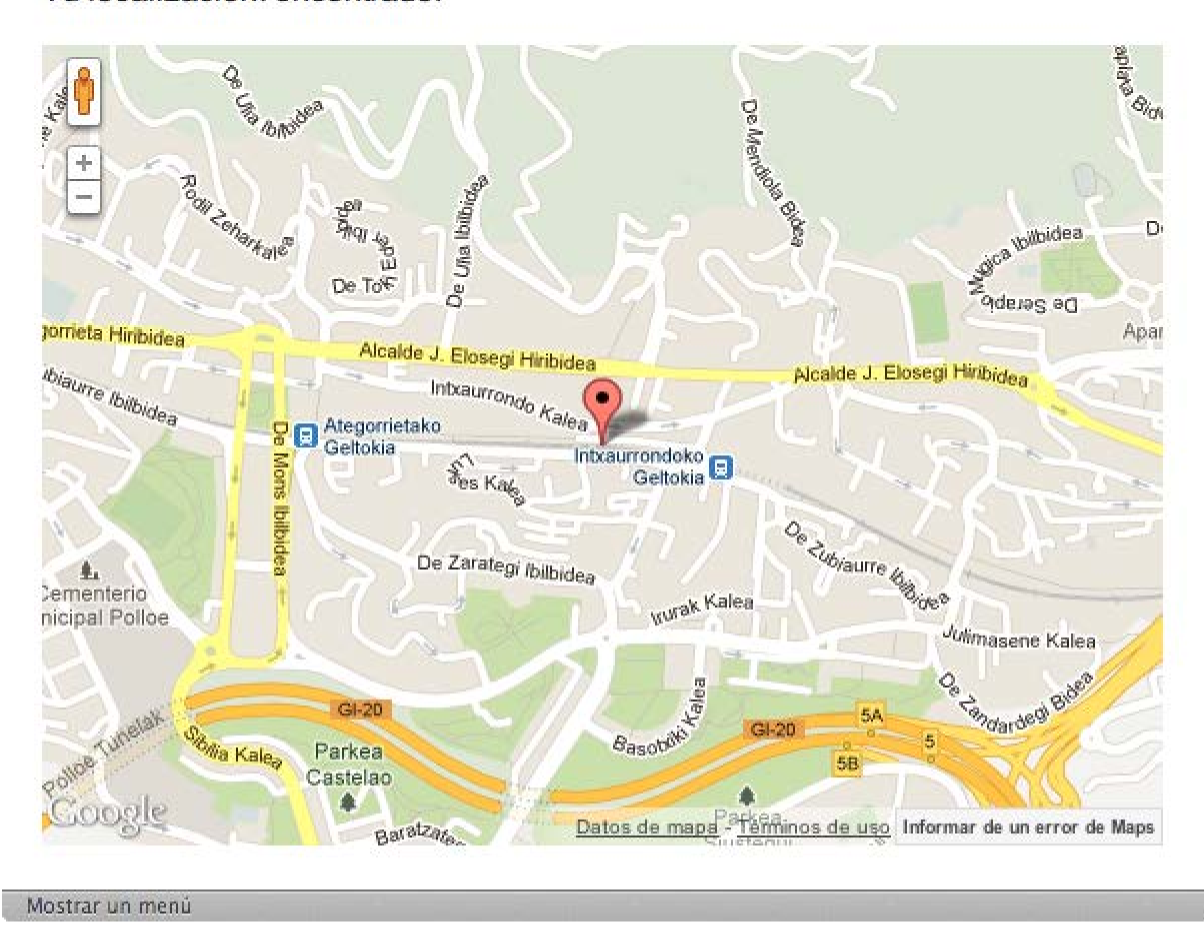


Figura 18.6 Posición actual en el mapa

18.11 CAPÍTULO 12

18.11.1 EJERCICIO 14

Crear un *Web worker* que dado un número entero, calcule todos los números primos comprendidos entre 1 y dicho número.

Proporcionaremos a este *Worker* un número entero, y devolverá un *array* con todos los números primos encontrados. Mostrar el listado de números primos en el documento principal.

 WebWorkers 

WebWorkers

Introduce un número entero mayor que 1: 1000 Calcular

Tiempo empleado: **1ms**

Primos: **2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997**

Mostrar un menú

Figura 18.7 Resultado tras calcular los números primos

18.12 CAPÍTULO 13

18.12.1 EJERCICIO 15

Implementar un servicio de chat, utilizando tecnología *Web Socket*, a partir del código proporcionado, que se describe a continuación.

- Se dispone de un servidor de chat, basado en Node.js, al cual deben conectarse los usuarios del chat. La dirección del servidor es `ws://www.arkaitzgarro.com:1337`.
- Se proporciona la interfaz básica (HTML y JavaScript) para desarrollar la aplicación. Es necesario implementar la conexión con el servidor y el intercambio de mensajes, así como la actualización de la interfaz.
- El protocolo de mensajes es el siguiente:
    - El primer mensaje a enviar al servidor es el *nick*. Tomad el valor de la caja de texto y enviarla al servidor como una cadena de texto.
    - Una vez estamos "registrados" en el servidor, éste puede comenzar a enviarnos mensajes, tanto los existentes hasta el momento, como los nuevos que se produzcan por otros usuarios. Estos mensajes (en formato JSON) corresponden con un objeto JavaScript, cuya estructura se muestra a continuación.
    - Finalmente, para enviar los mensajes de texto del usuario, enviamos la cadena

de texto directamente al servidor.

```
// Mensaje de tipo "color"
{
    type : "color",
    data : "" // Valor del color asignado por el servidor
}

// Mensaje de tipo "history"
{
    type : "history",
    // Un array que contiene los últimos mensajes enviados
    data : [{author, text, color, time}]
}

// Mensaje de tipo "message"
{
    type : "message",
    // Un objeto que contiene un mensaje enviado por otro usuario
    data : {author, text, color, time}
}
```

Descargar sitio web.

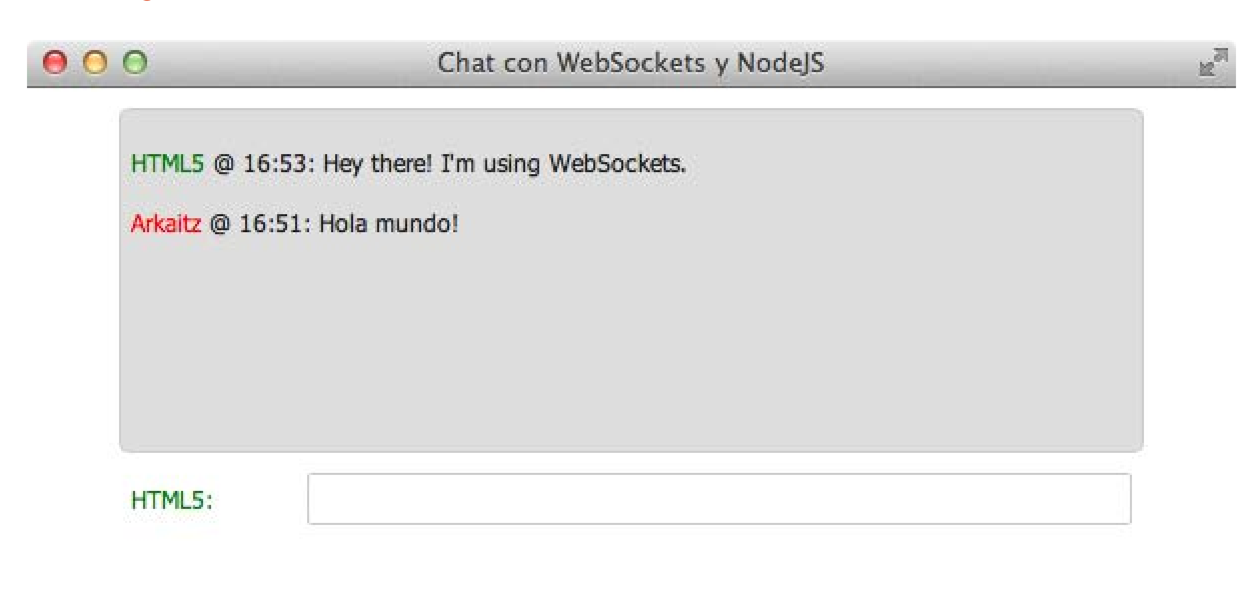


Figura 18.8 Resultado final del chat

ÍNDICE DE CONTENIDOS